

# Python.

## Part One: The Python Language

1.	Getting started .....	2
2.	Modules and packages .....	14
3.	Variables, simple values, and operators .....	18
4.	The mathematical library .....	25
5.	Strings .....	28
6.	Basic iterables: lists and tuples .....	40
7.	Operations only applicable to lists .....	46
8.	Sets .....	48
9.	Statements .....	50
10.	Documentation and help .....	55
11.	Assignments with patterns and pattern matching .....	56
12.	Formatting for strings and output .....	58
13.	Dictionaries .....	65
14.	enum types .....	68
15.	Functions .....	70
16.	Operators as functions .....	78
17.	Special operations on functions .....	80
18.	Bytes and bytearray .....	81
19.	Reading and writing files .....	86
20.	Classes .....	98
21.	Operators and printing for objects .....	104
22.	Special methods for classes .....	107
23.	Decorators .....	113
24.	Inheritance .....	117
25.	Iterables and iterators .....	122
26.	Generators .....	132
27.	Saving and restoring live data .....	136
28.	Exceptions: detecting and handling errors .....	143
29.	Dates and times .....	151
30.	Minor data structures .....	153
31.	Access to operating system features .....	158
32.	Multiple streams of execution - Threads .....	164
33.	Multiple streams of execution - Processes .....	176
34.	WWW services .....	182
35.	Network clients and servers .....	195
36.	Polling - asynchronous communication .....	201
37.	asyncio - more asynchronous communication .....	217

## Part Two: Graphics and User Interfaces

38.	Turtle .....	238
39.	Tkinter - the Canvas .....	247
40.	Universal widget methods .....	265
41.	Label .....	266
42.	Multiple items in a window .....	270
43.	Button .....	274
44.	Entry - simple text input .....	277
45.	Binding to mouse and keyboard events .....	280

46.	Text .....	282
47.	Scrolling .....	293
48.	Checkbutton .....	298
49.	Radiobutton .....	300
50.	Scale .....	301
51.	Listbox .....	303
52.	Spinbox .....	306
53.	Frames - windows within windows .....	309
54.	LabelFrame .....	311
55.	PanedWindows .....	312
56.	The Tk window .....	314
57.	Toplevel: an independent window .....	317
58.	Ttk widgets .....	317
59.	ttk.Progressbar .....	322
60.	ttk.Combobox .....	324
61.	ttk.Notebook .....	326
62.	ttk.Treeview .....	328
63.	Menus .....	336
64.	Menubutton and Optionmenu .....	342
65.	Dialogues .....	344
66.	Pillow - better image processing .....	351

# Part One: The Python Language

## 1. Getting started

Python is quick and easy to install on just about anything. The method varies a lot depending on your hardware and operating system. The best way for your set-up can be found by a google search for “install python 3”, I found that just clicking on “downloads” is not enough, I have to keep it pressed for a while, but others report different experiences.

The documentation is at <https://docs.python.org/3/>, but the documentation has many deficiencies, and the web may provide a hundred incompatible answers for any question you might have. That is the reason for these notes.

There are two ways to work with python. I find the best to be using the “IDLE” app which is completely interactive, you can type python code and get an immediate response as with a calculator but with more abilities, and you can still directly run larger programs stored in files.

The other way is to use your favourite text editor to write python code, save the file then double-click on its icon. That is a bit cleaner as no state/information survives from one run to the next, but it does not support interactive development which can be very helpful.

## i. Windows setup

Windows: `C:\Program Files\Python311\Lib\idlelib\idle.pyw` is where it was installed on my computer, but the start menu's search should be able to find it for you. The first time I installed Python, this file was a short-cut, and that was a useful thing. By default, Python will store and find all your programs in the same directory as all of its executables and other essentials, and that is not a good idea. To change it just right-click on the Idle shortcut's icon and select properties. Type the folder you want to use, such as `C:\python`, as the entry for "Start in", and it's done. More recently it was installed differently, and it took some work to set things up. There may be a better way, but this does work. First I made a copy of `idle.pyw` and changed its name so that it has a meaningless extension, I chose `idle.pys`. Then I made a short-cut to the new `idle.pys` and in its "properties" window made two changes. First in the general tab I set "opens with" to "`pythonw.exe`" (which the windows search thing was able to find), then in the shortcut tab I set "start in" to be the directory that I want my Python files to live in. That shortcut icon can be moved to a convenient place on the desktop, but for some reason windows won't allow it on the start bar.

When you're using Idle, `Alt+P` and `Alt+N` let you run up and down through previous commands that you have typed, perhaps modify them, then press `Enter` to re-execute them.

Idle gives you automatic indentation, which would be nice, but in the current version it is uncontrollable. Idle's options menu allows you to set the tab size, but every time you restart Idle it forgets. This didn't happen in the previous version, so I'm hoping it's just a temporary mistake that will be corrected soon.

## ii. Macintosh and Unix

On a Mac, I've got no idea what you do to change where it stores and looks for programs. Google searches produce many hits, but they are all wrong. To run up and down through previous commands you use `Control+P` and `Control+N` as Mac's have no `Alt` button.

We also have python on our server, `rabbit.eng.miami.edu`. The command to run it is just `python` (not `idle`). You use the keyboard up and down arrow keys instead of `Alt+P` and `Alt+N` to re-visit previous commands, and it doesn't give you automatic indentation. Type `control-D` or `exit()` to exit.

## iii. Starting to use Python

Python, through Idle, is interactive. If you type an expression it will be evaluated and printed. You can also define functions and classes and build up whole programs, but that isn't recommended for anything but smallish experiments: what you enter into Idle can not be saved usefully. "save as" just saves the entire transcript of everything you have done.

Here is a sample session that defines and uses two variables and a function

```

1 1 >>> x = 12 * 6
2 >>> y = x - 4
3 >>> def f(a, b):
4 ...     print(a, b)
5 ...     print("Done")
6 ...     return a + b
7 ...
8 >>> x
9     72
10 >>> y
11     68
12 >>> f(x, y)
13     72 68
14     Done
15     140

```

The >>> and ... business at the left is just Idle's prompt. Anything that appears on the line after either of those is something that I typed. Any line without a prompt was produced by python itself. After a function definition, you have to type an empty line as shown above. I will not bother to include that blank line in future examples.

print(...) takes any number of parameters and does its best to print them in an understandable way. The default behaviour is to print a single space between each item, and to end with a newline character. The defaults can be over-ridden:

```

2 1 >>> print(1, 2, 3, 4)
2     1 2 3 4
3 >>> print(1, 2, 3, 4, sep = "..."):
4     1...2...3...4
5 >>> print(1, 2, 3, 4, sep = "")
6     1234
7 >>> print(1, 2, 3, 4, sep = ", ", end = "")
8     1, 2, 3, 4

```

This last example would not have started a new line after printing the 4, but Idle always tidies up the last line of output by adding a new line if there wasn't already one there.

Note that correct and consistent indentation is absolutely required. While you are using Idle, it usually provides the necessary indentation for you.

However, a sequence of a few simple statements may be put all on one line with semi-colons to separate them. This is permitted, but generally not a good idea. Anyone reading Python will be familiar with the normal layout, a departure could easily cause a misreading.

```

3 1 >>> print(1, 2, 3, 4, sep = ", "); print(999)
2     1, 2, 3, 4
3     999
4 >>> print(1, 2, 3, 4, sep = ", ", end = ""); print(999)
5     1, 2, 3, 4999

```

#### iv. Loading and saving programs

When you are developing a program, there is nothing wrong with using your normal favourite editor, but if it doesn't support automatic indentation, you might get a bit annoyed. Idle provides its own editor that does do automatic indentation. From Idle's menu select file: open or file: new file. The name of the file should end in `.py` or `.py3`. After saving your work, go back to idle and type "import" followed by your file's name, without the `.py`. Idle's default indentation is too wide for my liking. If you want to change it, just use the options menu, then Configure Idle, and go to the windows tab. You can change a lot of other settings there too.

So, if I had saved the three definitions (sample 1) above as "program.py", I could do this.

```
4 1 >>> import program
  2 >>> program.y
  3     68
  4 >>> program.f(10, program.y):
  5     10 68
  6     Done
  7     78
```

If you have not managed to make Idle use a sensible directory for your `.py` files, you can enter these two lines:

```
5 1 >>> import os
  2 >>> os.chdir("C:\\python\\programs")
```

This will have no effect on anything you do from the file menu, but it will set where Idle looks for imports and other files that your program opens. It is annoying to have to type them every time you start a new session, there really should be some way to make things happen automatically.

If you use Idle's own editor, it has a run menu which contains a run module command. That will reset your Idle session, automatically import your file and run it. That sounds convenient, but it has problems. When you are developing or debugging a program, you will want to build up some reusable tests without having to build them into your program.

If you change your program, you can not use `import` again to load the new version. Once something has been *successfully* imported, Python ignores subsequent imports of the same file in order to save memory. This is what you need:

```
5 1 >>> from importlib import reload
  2 >>> reload(program)
  3     <module 'program' from 'C:\\.....\\program.py'>
```

In this form, `from` only imports specific things from a file, but it imports them into the standard "namespace". That means you don't have to type "importlib.reload".

When you import (or reload) a `.py` file, it is *almost* exactly the same as it would be if you typed the contents of the file directly into idle. The only big difference is that if an expression appears alone on a line, it is still executed, but the printing of the result is suppressed. So if you have a line that just says `12*9`, you will not see the `108` as you normally would, but things like prints still happen.

Another way to execute a `.py` file is to just double-click on it. So long as the extension `.py` (or `.py3`) is properly associated with the `python.exe` app (not `pythonw.exe`), a window will appear and your program's results will be displayed in it. But as soon as the program finishes, the window disappears and you don't have time to see what it said. The way to fix that is to make sure that your file's last line just says

```
input()
```

The `input` function waits until the user enters a line. That could just mean pressing ENTER. The `input` function returns whatever the user typed as its result. If you give it a parameter, that will be used as the prompt.

```
1 >>> def thing():
2 ...     x = input("Type something: ")
3 ...     print("You said", x)
4 ...
5 >>> thing()
6     Type something: one two three
7     You said one two three
```

Comments begin with a `#` sign, but also strings appearing alone are ignored so they can be used as comments.

```
1 >>> # this is a comment, nothing will happen
2 >>> "This will also be ignored."
```

As python programs are not compiled, may errors inside function definitions are not detected until the function is called and the defective code is executed. This can be quite annoying.

## v. At last, actually doing something with Python

This section just presents the basics needed to do anything useful. Everything is expanded in detail in later sections. The best way to start with Python is to just try it out. Start Idle and type things. This section only covers the most basic things necessary for doing anything useful. There is much more, all covered in later sections.

### Numbers

Python supports a few different kinds of number, here we will only look at two: `ints` (whole numbers) and `floats` (with decimal points). People will tell you that python is a typeless language, but that is not strictly true. It has types, but does everything related to types automatically. You do not have to declare variables (in

fact you can't in any meaningful way), just type a name follow it by an = and then a value. Python can tell the type of any value it sees and does obvious conversions as needed. One variable can hold values of different types at different times. This is something of a convenience, but quite dangerous and a source of many errors that would otherwise be easily preventable.

The operators are generally the ones you are used to if you have previously programmed in C, C++, or Java:

+ \* - / % == != < > <= >=

Their priorities are as expected and ( parentheses ) do their normal job. There is also a to-the-power-of operator \*\*.

```
1 >>> 2 * 3 + 4 * 5
2      26
3 >>> 2.1 * 3.2 + 4.007 * 5.13
4      27.2759099999999996
5 >>> 2 * 6.31
6      12.62
7 >>> 2 < 3
8      True
9 >>> 2 ** 8
10     256
```

As you see, the logical values are True and False, first letter capital is compulsory. The logical operators are spelled out as words.

```
1 >>> 2 < 3 and 4 < 5
2      True
3 >>> 2 < 3 and 4 > 5
4      False
5 >>> True and True or False and False
6      True
7 >>> True and (True or False) and False
8      False
9 >>> not 3 < 4
10     False
11 >>> (not 3) < 4
12     True
```

The last example illustrates one of the many dangers of being lackadaisical with types. The expression (not 3) < 4 has no sensible meaning but Python just doesn't care. not 3? The logical negation of a number? not 3 turns out to be False. Then False < 4 is also meaningless but still comes out to be True.

Division is a little bit different in Python. For the other arithmetical operations two ints always give an int result, and two floats or a mixed int and float always produce a float result.

/ always produces a float result regardless of its operands and its counterpart // always produces an int result regardless of its operands.

## Strings

A string can be any sequence of characters (with a few exceptions) surrounded by quotes. The quotes may be "double" or 'single', it makes no difference, except

that a string begun with one kind of quote must be ended with the same kind of quote. Inside double quotes a single quote is nothing special and vice versa, inside single quotes a double quote is nothing special. There is no concept of a character type, a single character is just a string of length 1. Strings may be added together or multiplied by a number:

```
1 >>> "One two three"
2      'One two three'
3 >>> 'One two three'
4      'One two three'
5 >>> "cat" + 'tle'
6      'cattle'
7 >>> "I said 'hello'"
8      "I said 'hello'"
9 >>> "It's raining"
10     "It's raining"
11 >>> "*" * 20
12     '*****'
```

A lot of things can be done with strings, here are just a few of the obvious ones:

```
1 >>> len("horse")
2      5
3 >>> "abcdefghijklmnopqrstuvwxy"[10]
4      'k'
5 >>> "abcdefghijklmnopqrstuvwxy"[10 : 14]
6      'klmn'
7 >>> "abcdefghijklmnopqrstuvwxy"[10:]
8      'klmnopqrstuvwxy'
9 >>> "horse" in "abcdefghijklmnopqrstuvwxy"
10     False
11 >>> "ghij" in "abcdefghijklmnopqrstuvwxy"
12     True
13 >>> "abcdefghijklmnopqrstuvwxy"[:10]
14     'abcdefghij'
15 >>> "horse" < "cat"
16     False
17 >>> "Horse" < "cat"
18     True
19 >>> "abðzЯ"
20     'abðzЯ'
```

Strings use Unicode rather than ASCII for their internal representation so they can contain all sorts of symbols if you can find a way to type them, but for "normal" characters it is as though ASCII were used.

Programmers can demand type conversions when the desired conversion is not automatic:

```
1 >>> float(23)
2      23.0
3 >>> int(3.95)
4      3
5 >>> str(3.95)
6      '3.95'
7 >>> float("12.34")
8      12.34
9 >>> int("987")
10     987
```



```
11 >>> type(12.3)
12     <class 'float'>
```

## Collections

A comma-separated list of values may be surrounded by square brackets to produce a list, or round brackets to produce a "tuple". The two are pretty much the same except that lists are "mutable" and tuples are not. If something is mutable, the values in it can be changed.

```
1 >>> a = [ 1, 2, "hello", True, 6 * 7, 3 > 12 / 6, "xxx" ]
2 >>> a[2]
3     'hello'
4 >>> a[-1]
5     'xxx'
6 >>> a[3]
7     True
8 >>> a[-2]
9     True
10 >>> a[-3]
11     42
12 >>> len(a)
13     7
14 >>> a + [ 100, 200 ]
15     [1, 2, 'hello', True, 42, True, 'xxx', 100, 200]
16 >>> a
17     [1, 2, 'hello', True, 42, True, 'xxx']
18 >>> a[2 : -2]
19     ['hello', True, 42]
20 >>> [1, 2, 3] * 4
21     [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
22 >>> a[5] = "hippopotamus"
23 >>> a
24     [1, 2, 'hello', True, 42, 'hippopotamus', 'xxx']
25 >>> b = [99, 88, 77, [666, 555, 444], 33, 22]
26 >>> b[3]
27     [666, 555, 444]
28 >>> b[3][1]
29     555
30 >>> c = (1, 2, "pony")
31 >>> c[1]
32     2
33 >>> c + b
34     Traceback (most recent call last):
35         File "<pyshell#137>", line 1, in <module>
36             c + b
37     TypeError: can only concatenate tuple (not "list") to tuple
38 >>> c + tuple(b)
39     (1, 2, 'pony', 99, 88, 77, [666, 555, 444], 33, 22)
40 >>> c[1] = "different"
41     Traceback (most recent call last):
42         File "<pyshell#139>", line 1, in <module>
43             c[1] = "different"
44     TypeError: 'tuple' object does not support item assignment
```

If something is immutable and you want to make a change, you have to make a copy of it with the changes built in from the beginning. In the following, the strange notation (123456789, ) is used to create a tuple of length 1. Parentheses

alone would be interpreted in the normal mathematical way, and as there are no operators involved they would have no effect. The comma makes it explicit that this is to be taken as a tuple.

```
1 >>> c = (1, 2, "pony", 3.14, 6, 99, 0)
2 >>> d = c[0 : 2] + (123456789, ) + c[3:]
3 >>> d
4      (1, 2, 123456789, 3.14, 6, 99, 0)
```

## Statements

The assignment statement has already been seen in its most basic form, and there are extensions (some expected, some not) too. The new value of a variable may be based on the current:

```
a = 3 * a
```

the usual update assignments are available:

```
a += 1
```

```
c *= 4
```

multiple assignments may be strung together, so this sets all of a, b, and c to 999.

```
a = b = c = 999
```

multiple assignments are also possible by assigning a tuple (or list) of values to a tuple of variables, "unpacking" a structure:

```
1 >>> x = (11, (22, 33))
2 >>> (a, (b, c)) = x
3 >>> a
4      11
5 >>> b
6      22
7 >>> c
8      33
```

In this form of assignment, the individual assignments act as though they happen simultaneously rather than sequentially, so `(a, b) = (b, a)` is a valid `swap(a, b)` which incidentally could not be written in Python, as there is nothing like reference variables or pointers in the C and C++ style.

```
1 >>> a
2      11
3 >>> b
4      22
5 >>> c
6      33
7 >>> (a, b, c) = (b, c, a)
8 >>> a
9      22
10 >>> b
11     33
12 >>> c
13     11
```

If statements behave in the familiar way, but in a generally unfamiliar syntax. The `if` and its condition appear on one line, ending with a colon, the statements to be executed if the condition is `True` appear on separate lines after that, and they must be indented to a consistent level deeper than that of the `if`. Then there is an optional `else:` that must have the same indentation as the `if` that it matches, and is followed by the statements to be executed if the condition is `False`,

indented in the same way as for the `True` case. Before the `else` (if there is one) there may be any number of `elif`s (for `else if`) followed by their own condition and colon:

```
1 >>> if x < 0:
2 ...     print("it is negative")
3 ...     x = - x
4 ... elif x == 0:
5 ...     print("it is zero")
6 ...     zero_seen = True
7 ...     count += 1
8 ... else:
9 ...     print("it is positive")
10 ...    count += 1
11 ...
```

inside a structured statement of this kind, `idle` gives a `...` prompt as an indication that it expects more. At the end, you must type a blank line or it will keep on expecting more. Nested `ifs` (or other structured statements) require uniformly increased indentation as would be expected.

The simplest kind of loop is the familiar `while`. It is given a condition (followed by a colon) and any number of statements to be re-executed for as long as the expression remains `True`. `break` and `continue` statements are allowed in loops and have the usual effect. In addition to all of this, a `while` loop may have an `else:` clause just like an `if`. The `else` statements are executed after the loop terminates, but only if it terminated naturally (the condition became `False`), not if a `break` statement caused the end.

```
1 >>> k = 1
2 >>> while k < 100:
3 ...     if k % 10 == 7:
4 ...         print("Emergency! ", k, "'s last digit is seven!", sep = "")
5 ...         break
6 ...     k *= 2
7 ... else:
8 ...     print("k never ended in 7 throughout the whole loop")
9 ...
```

## Ranges

A range is an example of an unusual kind of construct that Python has many more of.

If `a` and `b` are ints, `range(a, b)` produces a list of all ints starting with `a`, but ending *just before* it reaches `b`. `range(a)` is the same as `range(0, a)`. `range(a, b, c)` is the same as `range(a, b)` except that the numbers increase in steps of `c` instead of steps of 1. But there is a twist here. One of the purposes of a range is in controlling a loop, and loops can have very large ranges. So a range is never evaluated more than it has to be at any given moment. If I type `range(3, 7)` the response is the same, `range(3, 7)`, because it was never used so there was no reason to evaluate it. To see the whole list it must be forcibly converted into a list (or tuple). Indexing a range (with square brackets) only produces the element(s) needed.

```

1 >>> range(3, 10)
2     range(3, 10)
3 >>> list(range(3, 10))
4     [3, 4, 5, 6, 7, 8, 9]
5 >>> tuple(range(3, 10))
6     (3, 4, 5, 6, 7, 8, 9)
7 >>> range(3, 10)[2]
8     5
9 >>> range(3, 10)[2 : 3]
10    range(5, 6)
11 >>> list(range(3, 10)[2 : 5])
12    [5, 6, 7]

```

Ranges, lists, tuples, and other range-like things yet to be seen may control for loops, which have a very restricted syntax: no initialisation, no condition, no update, just an ordered collection of values to iterate over:

```

1 >>> count = 0
2 >>> for i in [4, 7, 12, "hello", 9]:
3 ...     count += 1
4 ...     print("item", i)
5 ... else:
6 ...     print(count, "uninterrupted items")
7     item 4
8     item 7
9     item 12
10    item hello
11    item 9
12    5 uninterrupted items
13 >>> for i in range(3, 10):
14 ...     print(i, end = " ")
15    3 4 5 6 7 8 9
16 >>> for i in "Hippopotamusses":
17 ...     print(i, end = " - ")
18    H - i - p - p - o - p - o - t - a - m - u - s - s - e - s -

```

## Comprehensions

Comprehensions allow a list (or list-like thing) to be made by applying a given operation to every member of another list-like thing. The syntax can probably be best understood just by seeing it. The first example takes a list of numbers [3, 7, 4, 2, 6] and multiplies each by ten and adds one to make another list of numbers:

```

1 >>> [ 10 * i + 1 for i in [3, 7, 4, 2, 6] ]
2     [31, 71, 41, 21, 61]
3 >>> [ 10 * i + 1 for i in (3, 7, 4, 2, 6) ]
4     [31, 71, 41, 21, 61]
5 >>> ( 10 * i + 1 for i in [3, 7, 4, 2, 6] )
6     <generator object <genexpr> at 0x0000023641DAF2A0>
7 >>> tuple(( 10 * i + 1 for i in [3, 7, 4, 2, 6] ))
8     (31, 71, 41, 21, 61)

```

Naturally, comprehensions may be nested:

```

1 >>> [ j / 10 for j in ( 10 * i + 1 for i in [3, 7, 4, 2, 6] ) ]
2     [3.1, 7.1, 4.1, 2.1, 6.1]
3 >>> names = [ (1, "one"), (2, "two"), (3, "three"), (4, "four") ]
4 >>> [ (b, "is", a) for (a, b) in names ]
5     [('one', 'is', 1), ('two', 'is', 2), ('three', 'is', 3), ('four', 'is', 4)]

```

```

6 >>> [ list() for n in range(1, 7) ]
7      [[], [], [], [], [], []]
8 >>> [ list(range(n, 7 * n, n)) for n in range(1, 7) ]
9      [[1, 2, 3, 4, 5, 6],
10     [2, 4, 6, 8, 10, 12],
11     [3, 6, 9, 12, 15, 18],
12     [4, 8, 12, 16, 20, 24],
13     [5, 10, 15, 20, 25, 30],
14     [6, 12, 18, 24, 30, 36]]

```

I reformatted the last output to make it more clearly what it is, a multiplication table.

Comprehensions may also be filtered. This last example starts with all the multiples of 3 that are less than 50, discards any that are not divisible by 2, and multiplies the rest by 10 to produce a new list:

```

1 >>> [ i * 10 for i in range(0, 50, 3) if i % 2 == 0 ]
2      [0, 60, 120, 180, 240, 300, 360, 420, 480]

```

## Functions

A function definition can be very simple: just the word `def` to introduce it, then its name, then a comma-separated list of parameter names in parentheses, and the usual colon, all followed by the statements that make up the function with the expected extra indentation. `return` statements with optional values may also appear. But be aware that there are a lot of more sophisticated and complicated things still to come. Recursion is supported as usual:

```

1 >>> def factorial(n):
2 ...     f = 1
3 ...     i = 2
4 ...     while i <= n:
5 ...         f *= i
6 ...         i += 1
7 ...     return f
8 ...
9 >>> factorial(7)
10    5040
11 >>>
12 >>> def factorial(n):
13 ...     if n == 0:
14 ...         return 1
15 ...     return n * factorial(n - 1)
16 >>> factorial(7)
17    5040

```

It must be kept in mind that Python does not have declarations in the conventional sense. A function definition is really an assignment of a value (an executable object made from the statements) to a variable (the name of the function), and a function is not defined until its `def` statement is executed. If a second `def` or an assignment to the function's name is executed, the function will change. One effect of this is that function definitions may be conditional:

```

1 >>> if a < 10:
2 ...     def f(x):
3 ...         return 99 * x
4 ... else:

```

```

5 ...     def f(x, y):
6 ...         return 100 * x + y
7 ...
8 >>> f(10)
9         990
10 >>> a = 12
11 >>> f(10)
12         990
13 >>> g = f
14 >>> g(100)
15         9900

```

Lines 10 to 12 just illustrate that there is no magic involved.

Functions do not need names. A lambda expression turns an expression and parameter list into a function that can be used in the normal way. The first example does just incidentally give the function a name by saving it in a variable, but that is just to make the introduction gentler than it would otherwise be. In the second example, `add_up_results` is an ordinary named function, it is the lambda that it is called with that is of interest (the sum of the squares of the numbers 1 to 5 is indeed 55). The third example makes a function that is aware of its environment at birth.

```

1 >>> f = lambda x, y: x * 100 + y
2 >>> f(5, 7)
3         507
4
5 >>> def add_up_results(f, a):
6 ...     total = 0
7 ...     for i in range(len(a)):
8 ...         total += f(a[i])
9 ...     return total
10 >>> add_up_results(lambda x: x * x, [1, 2, 3, 4, 5])
11         55
12
13 >>> def multiplier(x):
14 ...     return lambda y: y * x
15 >>> m = multiplier(6)
16 >>> m(7)
17         42
18 >>> m(9)
19         54

```

## 2. Modules and packages

It may be a bit early to be talking about modules and packages, but without some information there are things that won't make much sense.

### i. Modules

“Module” usually just means a `.py` file. A lot like what we would normally think of as a program, but they aren't really programs in the normal sense. They are

completely unstructured, just a collection of variable definitions, function definitions, class definitions and even executable statements.

When you say `import name`, Python will search for a file called `name.py` or possibly `name.py3`. First it searches its internal memory, called the cache: maybe you've already imported this module and it can save some time. Then it searches all of the directories listed in your path. Usually your path will begin with an empty string which represents whatever is considered to be the current working directory. Next it has the directory that you chose if you followed the steps in section 1 and set Idle's "start in". Then there will be a list of directories that the official Python libraries were stored in at installation. Python is quite flexible, sometimes you will even see `.zip` files as well as directories in your path. To see your path and maybe add another directory to it, do this:

```
1 >>> import sys
2 >>> sys.path
3     ['', 'D:\\python', 'C:\\Programs\\Python\\Python311 ... ]
4 >>> sys.path = sys.path[0:2] + [ "D:\\abc" ] + sys.path[2:]
```

That last step would insert the directory `D:\abc` as the third entry. Don't worry about the `[0:2]` and `[2:]`, they will become clear very soon in the section on lists and tuples.

When you import a module, you refer to the things it contains by putting the module name and a dot in front of the thing's name, as we saw before.

```
1 >>> import math
2 >>> math.factorial(7)
3     5040
4 >>> math.sqrt(math.pi)
5     1.7724538509055159
```

If a module has a long name, this can become annoying. There is an alternative that allows you to provide an abbreviation.

```
1 >>> import math as m
2 >>> m.factorial(7)
3     5040
4 >>> m.sqrt(m.pi)
5     1.7724538509055159
```

And remember that if you edit a module that has already been imported without errors, you can not import it again, you must use the `reload` function from `importlib`. If you imported something with an abbreviated name, you must use that abbreviation when reloading, so if you were brave or foolish enough to edit the `math` library, then in the case of the last example you would enter `reload(m)`, but in the one before that, `reload(math)`.

If there are just a few functions in a library that you want to use a lot, there is another form that allows you to leave out the module name and dot altogether.

```
1 >>> from math import sqrt, pi, factorial
```

```

2 >>> factorial(7)
3     5040
4 >>> sqrt(pi)
5     1.7724538509055159

```

You can even import everything from a library in that way with `from math import *`, but that is not recommended. Some libraries define a very large number of things, some with names you would never guess at. They can cause confusing conflicts with names in your own program.

Sometimes you want a file to behave differently when it is being imported than when it is being run directly (perhaps through being double-clicked on). This test will let you tell the difference (the long lines `__` are double underlines):

```

1 >>> if __name__ == "__main__":
2 ...     print("Being run directly")
3 ... else:
4 ...     print("Being imported")

```

## ii. Packages

A package is nothing more than a collection of modules. It will usually be a directory (folder) containing some `.py` files, and maybe some subdirectories also containing `.py` files and sub-sub-directories and so on. To make a directory into a package, it must contain an empty file called `__init__.py`. The structure we'll be using for this example is:

```

In my normal Python working directory, D:\python, there is
  a directory called mcpackage which contains
    an empty file called __init__.py
    a file called one.py
    a file called two.py
    a directory called junior which contains
      an empty file called __init__.py
      a file called three.py
      a file called four.py

```

The contents of the four non-empty files are:

```

D:\python\mcpackage\one.py:
def square(x):
    return x * x

```

```

oneval = 111

```

```

D:\python\mcpackage\two.py:
def cube(x):
    return x * x * x

```

```

twoval = 222

```



```
D:\python\mcpackage\junior\three.py:
def halve(x):
    return x / 2

threeval = 333
```

```
D:\python\mcpackage\junior\four.py:
def printtwice(x):
    print x
    print x

fourval = 444
```

This shows how to access everything

```
1 >>> import mcpackage.one
2 >>> import mcpackage.junior.four
3 >>> mcpackage.one.oneval
4     111
5 >>> mcpackage.junior.four.printtwice(mcpackage.one.oneval)
6     111
7     111
```

Which is all very clumsy. So now we will edit the two `__init__.py` files. To allow users to say "from ... import \*" to a package, that package's `__init__.py` must define a variable called `__all__` to be a list of the names of the things \* will select. Anything not in the list is ignored. So ...

```
D:\python\mcpackage\__init__.py:
__all__ = [ "one", "two", "junior" ]

D:\python\mcpackage\junior\__init__.py:
__all__ = [ "three", "four" ]
```

Now things are a bit less annoying:

```
1 >>> from mcpackage import *
2 >>> from mcpackage.junior import *
3 >>> three.threeval
4     333
5 >>> one.square(four.fourval)
6     197136
```

Mysteriously, `from three import *` doesn't produce any errors, but it doesn't do anything either.

If there is any general initialisation that you want to be performed when a package is loaded, just put it in that package's `__init__.py`.

### iii. Installing extra modules

Some useful functionality is provided by modules that do not automatically come with Python. If you want one, you're going to have to install it yourself. I have never needed to do this under Unix, but there are some very popular extra modules such as `numpy` for numeric and especially matrix work. On the other hand, quite a few normal Python things don't work under Windows, so extra modules were needed there.

Be very careful. The main purpose of third party software seems to be to make it easier for viruses to spread. Check that a lot of people have been successfully using something before installing it.

We'll use the example of installing `pynput`. It gives special control over the keyboard and mouse. To start, you need a Python utility named `pip`, and getting and installing that is quite a nuisance, so check first to make sure you haven't already got it, the instructions below give hints at what to do and where to look. First you have to open a terminal (dos shell or whatever) and navigate with `cd` commands to the folder that contains your Python executables. Finding that folder is very easy: from within Python/Idle import the `sys` module and look at its `executable` attribute:

```
1 >>> import sys
2 >>> sys.executable      # under Windows
3     'C:\\Users\\ ... Python\\Python311\\pythonw.exe'
4 >>> sys.executable      # under Unix
5     '/usr/local/bin/python'
```

Then you need to download this link <https://bootstrap.pypa.io/get-pip.py> into that folder. The command for that is

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

Once you have done that, give the command

```
.\python get-pip.py
```

Pay attention to what it says. It told me that `pip.exe` had been installed into the `\Scripts` subdirectory, which is not on the path, so instead of using the command `pip`, I have to use the command `Scripts\pip`. I imagine that something similar will happen for you. Use this command

```
Scripts\pip --version
```

to verify installation. Finally, this command installs the `pynput` module.

```
Scripts\pip install pynput
```

After that, you shouldn't have to do anything else. Go to your Idle session and make sure `import pynput` works. If not, you may have to restart Idle. I didn't, but it could happen.

### 3. Variables, simple values, and operators

Python directly supports two kinds of numbers: whole numbers (`ints`) and numbers with decimal points (`floats`). If you import `complex`, complex numbers

are also available. The programmer almost never has to do anything about types, they are all handled automatically.

You create a variable just by assigning a value to it. No declarations. The rules for legal variable names are the same as in most programming languages. You can store any value at all in any variable you want. You don't even have to be consistent, you can take a variable that previously held an int, and store a string in it instead.

## i. Numbers and types

To type an int value, optionally type a + or - sign, and follow it with any number of digits, except that leading zeros are forbidden. Between digits, you may insert underline characters to make big numbers more readable. Underlines are like commas in the normal human writing of large numbers. `1_234_567` is one million two hundred and thirty-four thousand five hundred and sixty-seven. Beware: python lets you put underlines anywhere in a number. If you type `1_2345` by mistake, it will be accepted as `12345` without warning. An int can have any number of digits, they just grow as needed. Sometimes you need to take care of that, a very big int can occupy a lot of memory.

Int values may be preceded by `0x` for hexadecimal, `0b` for binary, or `0o` for octal.

```
1 >>> 0b1001
2      9
3 >>> 0xFF
4      255
```

To type a float value, do the same as you would with an int, but then you must either put a decimal point somewhere (`123.45`, `.123`, `345.`), or use the `e` notation (for times-ten-to-the-power-of), or both. Unlike ints, floats do not have unlimited precision, they are limited to the implementation given by your CPU. That usually means about 15 or 16 decimal digits of accuracy.

You can convert between types with a typecast. Typecasts look like functions.

```
1 >>> float(123)
2      123.0
3 >>> int(3.14)
4      3
5 >>> int(3.99999)
6      3
```

Python always displays a decimal point when it prints a float unless you explicitly tell it not to, see the section on formatting.

To create a variable or store a value in a variable, just use an assignment: variable name then = then the new value. If the variable doesn't already exist, it is created right there. No declarations, no need for a type.

Assignments may be strung together as in `a = b = c = 9`, which sets all three variables to 9.

You can find the type of anything by looking at the result of the `type` function. When the `type` function displays its result, it has a complicated format, but when you want to check a type it is much simpler:

```
1 >>> x = 99
2 >>> y = 23.
3 >>> type(x)
4     <class, 'int'>
5 >>> type(3.14159)
6     <class, 'float'>
7 >>> type(y) == float
8     True
9 >>> type(3 * 7) == float
10    False
```

From that, you see an example of another type. The values in the `bool` type are `True` and `False`. There is also another special value `None`. It is a bit like `NULL` or `null` or `nil` in familiar languages, it is used to say “no data here”. The type of `None` does not behave as we might expect:

```
1 >>> type(None)
2     <class 'NoneType'>
3 >>> type(None) == NoneType
4     Traceback ..... name 'NoneType' is not defined
```

but `None` is the only value in `NoneType`, so if you want to know whether something is a `NoneType` or not, just `==` compare it with `None`.

## ii. Very basic functions

We have not covered function definitions yet, but even so, the small ones that are about to appear should still be understandable. The keyword `def` begins a function definition, and the rest is quite straightforward.

If a variable is created outside of any function definition (i.e. with no indentation), it naturally becomes a global variable. If you use a variable inside a function, it is assumed to be a local variable (usually. There are exceptions, but it is not a good idea to take advantage of them). If a function needs to use a global variable, it should first explicitly state that the variable is global, as shown in these two examples:

```
1 >>> g = 123
2 >>> def f(x):
3 ...     # I say nothing so g is local
4 ...     g = x * x
5 ...     print("g =", g)
6 ...
7 >>> f(7)
8     g = 49
9 >>>

10     g
11     123
1 >>> g = 123
2 >>> def f(x):
3 ...     global g
4 ...     g = x * x
5 ...     print("g =", g)
6 ...
7 >>> f(7)
```

```
8      g = 49
9 >>> g
```

```
10     49
```

### iii. Numeric operations

Python provides all the usual operators. + \* - / % == != < > <= >=.

The / operator always produces a float result.

The // also does a division but produces whole numbers as its result. Unlike with other languages, it always rounds down, not towards zero.

The % operator's behaviour is compatible with that of //, so -99 % 10 is 1, not -9.

\*\* does to-the-power-of.

Comparisons may be strung together without the need of ands, e.g. 3 < x < 7.

There are also update assignments, += -= \*= /= //= \*\*= %= &= |= ^= >>= <<=. They all work the same way. So for one example, x \*= 3 is exactly the same as x = x \* 3, but you don't have to worry about parentheses: x \*= a + b is the same as x = x \* (a + b).

Due to the indentation rules, you can not spread a long expression over multiple lines unless there is something to make it clear that the expression hasn't ended yet, and a dangling operator isn't enough. If a line ends with "a \* b +" and the next line continues with "c \* d", that will be an error. If a line ends with "(a \* b +" and the next line continues with "c \* d)" everything is OK.

In an expression like -2, you are not typing the number "minus 2", you are typing the number 2 and applying the unary - operator to it. Due to operator precedence rules, this can have unexpected results, as seen in the last two examples below.

If you mix floats and ints (except for / and //) everything will be treated as floats.

```
1 >>> 1234567890 * 9876543210
2      121932263111263526900
3 >>> 12 * 120.0
4      1440.0
5 >>> 8 / 5
6      1.6
7 >>> 8 // 5
8      1
9 >>> -8 // 5
10     -2
11 >>> 3.6 // 2
12     1.0
13 >>> -3.6 // 2
14     -2.0
15 >>> 2 ** 5
16     32
17 >>> 2.1 ** 5.36
18     53.34520885174045
19 >>> -2 ** 0.5
20     -1.4142135623730951
21 >>> (-2) ** 0.5
```

22 (8.659560562354934e-17+1.4142135623730951j)

The last example illustrates that python understands complex numbers too. It uses the letter *j* instead of the mathematically usual *i*. You can type them just as *real + imaginaryj*. Python always surrounds complex numbers with parentheses, and leave no spaces around the + or -. You don't need to do that, but you must type the *j* right up against the imaginary part.

```
1 >>> y = 2 + 0.5j
2 >>> y
3      (2+0.5j)
4 >>> y = 2 + 0.5 j      # this causes an error
5 >>> 3j ** 2
6      (-9+0j)
```

In the first one, that is not just a single direct assignment from a complex number to a variable. First we get the ordinary int 2, then we get the complex number without a real part, 0.5j, then they are added to produce (2+0.5j) which is finally assigned to *y*.

If an expression gets very long, it should of course be spread over more than one line. Because Python uses line-breaks and indentation to determine the structure of a program, you can't just start a new line anywhere. There are two possibilities:

If any type of bracket (square, round, or curly) is still open at the end of a line, the next line will be taken as a continuation, so just put your long expression in parentheses.

If the last visible character on a line is \ the next line will be taken as a continuation. Naturally, the \ does not become part of the expression.

In the special case of a long string there is a third way. Use triple quotes to begin and end the string: `"""abc"""` or `'''abc'''`. Anything can go inside. The only way to end a triple-quoted string is with another `"""` or `'''` identical to the one that started it, or with the end of the file which would be an error.

#### iv. Logical and bit-wise operators

Every int has two useful methods. A method is like a function but different. To apply a method to something, you first type the something, then a dot, then the name of the method, and finally a pair of round brackets containing any extra parameters. When applying a method to an integer constant, there must be a space before the dot, otherwise it will be taken as a decimal point. `bit_length` tells you how many binary digits are needed to represent a number, and `bit_count` tells you how many of those binary digits are 1.

```
1 >>> 84 .bit_length()
2      7      # 84 in binary is 1010100, seven digits.
3 >>> 84 .bit_count()
4      3      # 84 in binary is 1010100, three ones.
```

`<<` and `>>` are binary shifts. Both “arithmetic”, not “logical”, only for ints.

```

1 >>> 1 << 8
2      256
3 >>> 256 >> 8
4      1
5 >>> -7 >> 1
6      -4

```

The logical operators are spelled out: `and` `or` `not`. Anything empty is considered to be false, anything non-empty is considered to be true. `0` and `None` are considered empty, as are empty data structures such as lists. `and` uses “short-circuit” evaluation: if the left operand is false, it is returned immediately; if the left operand is true, the right operand is returned. `or` also uses short-circuit evaluation: if the left operand is false, the right operand is returned; if the left operand is true, it is returned immediately.

```

1 >>> True and False
2      False
3 >>> 7 and True
4      7
5 >>> 0 and 1
6      0

```

The bitwise operators `&` `|` `^` `~` also exist, and do what they normally do. Except that `~` has surprising results. `8` in binary is `1000`, so one might expect `~8` to be `0111`, which is `7`, but python delivers `-9` as the result. This is because Python doesn't assume any particular bit size for ints, it acts as though the sign bit is repeated infinitely. So with `...` representing a long stream of identical digits, `8` is `...0000001000`, so `~8` is `...11111110111`, and that is the two's complement representation of `-9`.

The normal assignment operator, `=`, may not appear in expressions, but the operator `:=`, which does the same thing, may. Similarly `:=` may not be used as a statement.

```

1 >>> a = 66
2 >>> if (a := 9) > 7:
3 ...     print("yes")
4         yes
5 >>> a
6      9
7 >>> a := 4      # this causes an error

```

## v. Priorities

Operator priorities:

highest	(...)	parentheses
	x[...], x(...), x.abc	index, slice, call, attribute selection
	await x	coming later
	**	
	+x, -x, ~x	unary operators
	*, @, /, //, %	@ is recognised but has no set meaning

```

+, -
<<, >>
&
^
|
in, not in, is, is not, <, <=, >, >=, ==, !=
not x
and
or
x if y else z      see below
lambda            later, in the section on functions
lowest           :=

```

## vi. Other features

The form `x if y else z` allows conditional evaluation within a statement. When it is met, `y` is evaluated; if the value is considered true, then `x` is evaluated to produce the value of the whole expression, and `z` is never touched. If `y` comes out to be false, then `z` is evaluated to produce the value of the whole expression, and `x` is never touched.

```

1 >>> math.sqrt(0 if x < 0 else x)      # makes square root safe
2 >>> y = (a if a < b else b) + (c if c > d else d)
3                                     # adds the smallest of a and b to the largest of c and d
4 >>> print("Failed" if not ok else "Test was successful")
5                                     # which is equivalent to, but much less bulky than
6 >>> if not ok:
7 ...     print("Failed")
8 ...     else:
9 ...     print("Test was successful")

```

```

round(x)
    rounds x to the nearest integer
round(x, n)
    rounds x to n digits after the decimal point
abs(x)
    is for absolute value
min(a, b, c, d, e, ...)
    returns the minimum of all of its parameters
max(...)
    is as expected given min.

```

`any(x)` and `all(x)` take anything at all list- or tuple-like value for `x`, they examine all of the contents of `x`, seeing which are considered false (`False`, zeros, empty things) and which are considered true (all the rest). If even a single one is true, `any` returns `True`. If every one is true, `all` returns `True`. `any` says `True` about an empty `x`.



## 4. The mathematical library

import math provides five constants ...

```
math.pi
math.e
math.inf      the hardware representation of so called infinity
math.tau      which is 2 * math.pi
math.nan      which is the representation of something that should be a
              number but isn't. Hardware produces nan as the result of
              something like the square root of a negative, but Python catches
              those sorts of errors and throws exceptions (coming later)
              instead. It is very difficult to find any Python operation that
              produces a nan.
```

... and the common functions: acos, acosh, asin, asinh, atan, atan2, atanh, ceil, cos, cosh, exp, factorial, floor, gcd, isfinite, isinf, isnan, lcm, log, log10, log2, pow, sin, sinh, sqrt, tan, tanh, trunc, ...

```
1 >>> import math
2 >>> math.pi
3     3.141592653589793
4 >>> math.cos(math.pi / 4)
5     0.7071067811865476
```

... and some less common ones:

```
math.cbrt(x) is for cube roots.
math.comb(n, r) is the normal combinatoric function.
math.copysign(x, y) returns x, but changed to have the same sign as y.
math.degrees(r) converts from radians to degrees,
math.dist((x1, y1), (x2, y2)) is the Pythagorean distance, not limited to
two dimensions.
math.erf and math.erfc are the Gaussian error functions.
math.exp and math.exp2 are e-to-the-power-of and 2-to-the-power-of.
math.expm1(x) is (e to-the-power-of x) - 1, calculated very accurately.
math.fabs(x) is the same as abs(x) except that the result is always a float.
math.fmod is the % operator for floats. math.fmod(3.712, 0.5) = 0.212
because 3.712 is some whole number (7) * 0.5 + 0.212
math.frexp produces mantissa and exponent. frexp(8.625) = (0.5390625,
4) because 8.625 is 0.5390625 * 2 to-the-power-of 4. Mantissas
are always less than 1.
math.fsum adds together parameters always producing a float.
math.gamma is factorial, extended to work for floats. Actually, N! =  $\Gamma(N + 1)$ .
math.hypot(x, y) is sqrt(x*x + y*y) but not limited to two dimensions.
math.isclose(x, y, 0.0001) = are x and y within 0.0001 of each other?
math.isqrt(x) biggest int that is less than or equal to sqrt(x).
math.ldexp(x, y) = x * (2 to-the-power-of y). The opposite of math.frexp.
math.lgamma(x) = logarithm of abs of gamma
```

`math.log` is the natural (base e) logarithm  
`math.log1p(x) = math.log(1 + x)`, but much more accurate in special cases.  
`math.modf`: fractional and integer parts. `math.modf(73.185) = (0.185, 73.0)`  
`math.nextafter(x, y)` = the closest different float value to `x`, in the direction of `y`.  
`math.perm(n, r)` = number of permutations of `r` things taken from `n`. The one parameter form `perm(n)` is the same as `perm(n, n)`, which is pointlessly the same as `factorial`.  
`math.prod`, like `math.fsum` but multiplying.  
`math.radians(d)` converts from degrees to radians.  
`math.remainder(x, y)`, a bit like `x % y`, but it works for floats.  
`math.trunc(x)` - remove everything after the decimal point, round towards zero.  
`math.ulp(x)` is the difference between `x` and `math.nextafter(x)`.

## ii. Randomness

This is not part of the `math` library, but it seems to be the right place to cover it. The Python documentation warns that these methods are not good enough for cryptography. There is a separate module called `secrets` for that. `import random` and you get the following:

`random.random()`  
 delivers an evenly distributed random float somewhere between 0 and 1. 0 is possible, 1 is not.

`random.uniform(min, max)`  
 delivers an evenly distributed random float somewhere between `min` and `max`, which can of course be floats.

`random.gauss(mu, sigma)`  
 delivers a random number from a Gaussian or normal or bell curve distribution with mean = `mu`, default 0, and standard deviation = `sigma`, default 1.

`random.randint(min, max)`  
 delivers an evenly distributed random int somewhere between `min` and `max`. `min` and `max` are both possible results.

`random.randrange(stop)` and  
`random.randrange(start, stop)` and  
`random.randrange(start, stop, step)`  
 delivers a randomly selected int from the numbers that would be produced if `range(start, stop, step)` were called.

`random.getrandbits(nb)`  
 delivers an evenly distributed random positive nb-bit int. For example, `randbits(4)` is equivalent to `randint(0, 15)`.

`random.randbytes(nb)`  
 delivers `nb` evenly distributed random 8-bit values. The result is delivered as a bytes object, rather like a string (but no unicode), rather like a list of ints. Covered in a later section.

`random.shuffle(list)`  
randomises the order of the items in the list. Nothing is returned, the list itself is modified.

`random.choice(list)`  
delivers a randomly selected item from the list. `list` may in fact be anything that allows indexing with square brackets.

`random.choices(list, weights, k = 1)`  
delivers a randomly selected list of `k` items from the list. `k` is only accepted as a keyword parameter. All items are eligible for selection regardless of whether they have been selected before or not. If the `weights` parameter is provided, it should be a list with the same length as the first parameter. It specifies the relative likelihoods of each of the list's items being selected. They may be probabilities but any numbers will do. e.g. `[1, 3, 3]` means that `list[1]` and `list[2]` are equally likely, and both are three times more likely than `list[0]`.

`random.sample(list, k, counts)`  
delivers a randomly selected list of `k` items from the list, but without replacement. Once an item has been selected, it won't be selected again. `counts` is only accepted as a keyword parameter, if provided, it should be the same length as `list`. It specifies the number of times each item in `list` is duplicated to make the list that items are chosen from. e.g. `list = [1, 2, 3]` and `counts = [5, 3, 1]` is the same as just `list = [1, 1, 1, 1, 1, 2, 2, 2, 3]`. A particular case: `random.sample(list, len(list))` will produce a random permutation.

Random numbers may be selected from many more distributions than are listed here. If you really care about probability, check the official Python documentation.

Unlike most computer random number generation systems, Python does randomise the sequence automatically. In contrast, other systems start their random number generators in exactly the same state every time a program runs, so if the same program is run more than once, it will get the same sequence of random numbers. This is good for debugging, but bad for security and statistics if you forget about it. Python is the opposite. If you want reproducible behaviour, use `seed`, `getstate` and/or `setstate`.

`random.seed(n)`  
uses the number `n` to start or restart the random number generator. If you use the same seed again, you will get exactly the same sequence of random numbers. `n` can be an `int`, `float`, `str`, `bytes` or `bytearray`. If you don't provide a parameter, the current time is used instead, except that if your operating system provides a cryptographically good source of randomness, that will be used instead. To find out if that is provided, `import os`. `os.urandom(16)` will raise an exception if it isn't.

`random.getstate()` and  
`random.setstate(obj)`

is another way to get reproducible results. `getstate` returns an object which when given to `setstate` will return the random number generation system to exactly the same position as it is in now.

The documentation states that the `random` module is not good enough for cryptographic needs. For such purposes, use the `secrets` module instead. `secrets` has much less variety in its functions, but the essentials are there.

```
secrets.randbelow(n)
    Returns a random int between 0 and n-1 inclusive.
secrets.randbits(n)
    Generates exactly n random bits and returns the positive int that
    they represent.
secrets.choice(things)
    things can be anything reasonably list-like, iterators and
    generators are not included. Returns a randomly selected one of
    the things. All have equal probability.
secrets.token_hex(n)
    Generates n random bytes and returns them as a hexadecimal
    string.
secrets.token_bytes(n)
    Generates n random bytes and returns them as a bytes object.
```

## 5. Strings

Strings consist of (almost) any sequence of characters, even none at all, enclosed in quotes. Single quotes and double quotes are both allowed and mean exactly the same thing. Whatever type of quote you use to start a string, you must use the same type to end it. That makes it easy to print quotes. You can still use the normal `\` escape characters.

```
1 >>> "Hello"
2     'Hello'
3 >>> print("Hello")
4     Hello
5 >>> print('I said "Hello"')
6     I said "Hello"
7 >>> print("One\n\"Two\"")
8     One
9     "Two"
10 >>> "One\n\"Two\""
11     'One\n"Two"'
```

Notice that when a string is a result, python prints it with quotes. No matter what sort of quote you used, python will always use single ones, unless the string itself contains single quotes. But the `print` function is supposed to be more readable, so it doesn't show the quotes. Also, when a string is a result you see any newlines or other odd characters in it as `\n` or whatever, but when explicitly printed, the `\n`s are replaced by the characters they represent.

Strings are always referred to as strings, but their type is really just `str`:

```
if type(s) == str: ...
```

Newlines are not normally allowed inside quotes, but if you use triple quotes, newlines are allowed. That is also how you make long comments.

```
1 >>> def f(a, b):
2 ...     """
3 ...     This function adds two numbers together,
4 ...     I wrote it on a Monday evening.
5 ...     It is very good.
6 ...     """
7 ...     return a + b
```

Strings may also be spread over multiple lines just by surrounding a bunch of them in parentheses. No commas to separate them, that would make a tuple instead. The parentheses are only needed if the strings are spread over more than one line. The bunch of strings are all concatenated to make one big string. You don't even need to get the indentation right:

```
1 >>> ( "abc"
2 ...     "def"     "ghi"
3 ...     "jkl"
4 ...     "mno" )
5     'abcdefghijklmno'
```

Python has no type for single characters. A character is just a string of length 1.

The `str` function will turn just about anything into a string.

```
1 >>> str(3.14159)
2     '3.14159'
3 >>> str([45, "cat", [9, 8], False])
4     "[45, 'cat', [9, 8], False]"
```

And `int` and `float` will turn a string into the number that it looks like. `int` has an optional second parameter to specify the base. Spaces at the beginning or end of the string are ignored. If the string is not in the correct format, it will produce an error.

```
1 >>> float("356.71")
2     356.71
3 >>> float("2.76549e+4")
4     27654.9
5 >>> float("-12")
6     -12.0
7 >>> int("100")
8     100
9 >>> int("100", 8)
10    64
11 >>> int("123", 5)
12    38
```

```

13 >>> int("AAAAAAAAAAAAAAAAAAAAAAAAAAAA", 16)
14      886151997189943915269204706853563050

```

The `repr` and `ascii` functions do almost the same thing as `str`. `str` is what `print` uses and it is intended to produce nice human readable output. `repr` is what is used to display the value of an expression entered interactively, it is supposed to give more information for debugging and make it perfectly clear what type the object is. `ascii` produces a string in which all non-ASCII unicode characters are represented by their hexadecimal escape sequences. Unicode is the system used to represent special symbols and foreign letters that the ASCII code can't handle. All Python strings use unicode. More on that later in this section.

```

1 >>> x = fraction(3, 7)
2 >>> print(str(x))
3      3/7
4 >>> print(repr(x))
5      fraction(3, 7)
6 >>> print(ascii(x))
7      fraction(3, 7)
8 >>> print(str("abðΣzЯ"))
9      abðΣzЯ
10 >>> print(repr("abðΣzЯ"))
11      'abðΣzЯ'
12 >>> print(ascii("abðΣzЯ"))
13      'ab\u2202\u2211z\u042f'

```

`int("123")` and `float("3.14159")` convert strings into the numbers that they look like. spaces at the beginning or end are ignored.

Strings use unicode, not ASCII, to represent characters, so each character could occupy anything from 1 to 4 bytes, but the programmer doesn't need to worry about that. It is all abstracted away. Strings are sequences of characters, not sequences of bytes. Unicode means you can use strange symbols: "ø&Σ←ЯÆΩ". If you know the unicode index for a character in hexadecimal, you can include it in a string like this:

```

1 >>> "4\u00B2=16"
2      '4²=16'

```

B2 is the unicode position for a superscripted 2, as in 4 squared.

## ii. Immutability

Strings are Immutable. That means that once you have created a string, there is nothing you can do to change it. You can make new strings based on parts of an existing string to get the effect of a change, but nothing will change the original string.

```

1 >>> s = "abc"
2 >>> t = s
3 >>> t is s

```

```

4     True
5 >>> t += "def"
6 >>> t
7     'abcdef'
8 >>> s
9     'abc'
10 >>> s[1] = 'x' # anything like this will cause an error

```

The `is` operator is very different from the `==` operator. It does not look at an object at all, it just checks that they are in fact the *same* object. This is like comparing two pointers in many other languages.

The types `bytes` and `bytearray` both provide a kind of mutable string. They are covered in their own section later on.

### iii. Basic operations

Indexing a string gives you the character at that position (counting from zero), but remember it is really just a length-one string. `"abcdefghijkl"[4]` is `'e'`. Substrings, or slices, can also be taken. `s[i:j]` is the substring starting at position `i` and ending *just before* position `j`. `s[i:]` means go all the way from `i` to the end. Negative indexes count backwards from the end. `len` tells you the length of a string, it is an ordinary function, not a method.

```

1 >>> s = "abcdefghijklmnopqrstuvwxy"
2 >>> len(s)
3     26
4 >>> s[0]
5     'a'
6 >>> s[-3]
7     'x'
8 >>> s[9 : 12]
9     'jkl'
10 >>> s[3 : -3]
11     'defghijklmnopqrstuvw'
12 >>> s[20:]
13     'vwxyz'

```

`str + str` creates a new string consisting of the original two joined together. `str * int` creates a string that consists of a number of repetitions of the original.

```

1 >>> s + "0123456789"
2     "abcdefghijklmnopqrstuvwxy0123456789"
3 >>> "---" * 4
4     '-----'

```

With strings, the `in` and `not in` operators check for substrings.

```

1 >>> 'o' in "Hippopotamus"
2     True
3 >>> "pot" in "Hippopotamus"
4     True

```

```

5 >>> "pppp" not in "Hippopotamus"
6     True
7 >>> "Hippopotamus" in "Hippopotamus"
8     True
9 >>> "Hippopotamus" in "Hippo"
10    False

```

#### iv. Useful methods

`s.count(sub)` tells you the number of times a substring appears

```

1 >>> "The Cat Sat On The Mat 1234.".count("at")
2     3
3 >>> "The Cat Sat On The Mat 1234.".count("elephant")
4     0

```

The methods `.startswith` and `.endswith` do what their names suggest. `.removeprefix` and `.removesuffix` are similar: if `.startswith` or `.endswith` respectively is true, then a new string with that part removed is returned.

```

1 >>> s.startswith("abcd")    # at this point s is still the alphabet string
2     True
3 >>> s.startswith("horses")
4     False
5 >>> s.endswith('z')
6     True
7 >>> "item: cat".removeprefix("item: ")
8     'cat'
9 >>> "item: cat".removeprefix("other: ")
10    'item: cat'
11 >>> "project/program.py".removesuffix(".py")
12    'project/program'

```

If you've got a collection (list, etc) of strings, the `.join` method will join them all together.

```

1 >>> ", ".join(["abcd", "xy", "hello", "cat"])
2     "abcd, xy, hello, cat"

```

`s.find` tells you where a substring first appears in a larger string (-1 if it doesn't). `s.find` has two optional parameters, one telling it where to start the search, the other telling it where to stop. `s.rfind` does the same, but searches backwards from the end.

```

1 >>> s = "abcd979efghijkl979mnopq979rst"
2 >>> s.find("979")
3     4
4 >>> s.find("979", 5)
5     15
6 >>> s.find("979", 5, 8)
7     -1
8 >>> s.rfind("979")
9     23

```



```

10 >>> s[23]
11     '9'
12 >>> s.find("cat")
13     -1

```

`s.replace(sub, new)` replaces every occurrence of `sub` with `new`. It has an optional parameter to say the maximum number of replacements to be done. The method does not change `s`, it creates a totally new string.

```

1 >>> s = "abcd979efghijkl979mnopq979rst"
2 >>> s.replace("979", "!@#$$")
3     'abcd!@#$$efghijkl!@#$$mnopq!@#$$rst'
4 >>> s.replace("979", "!@#$$", 2)
5     'abcd!@#$$efghijkl!@#$$mnopq979rst'

```

`s.partition(sub)` looks for a substring, and returns three strings: everything before the first occurrence of `sub`, the `sub` itself, then everything after the first occurrence. `s.rpartition(sub)` looks for `sub` backwards from the end. `partition` behaves well in most cases when things are not as expected, see lines 6 to 10 below.

```

1 >>> s = "abcd979efghijkl979mnopq979rst"
2 >>> s.partition("979")
3     ('abcd', '979', 'efghijkl979mnopq979rst')
4 >>> s.rpartition("979")
5     ('abcd979efghijkl979mnopq', '979', 'rst')
6 >>> "12345".partition("x")
7     ('12345', '', '')
8 >>> "".partition("x")
9     ('', '', '')
10 >>> "abcd".partition("")           # this causes an error

```

`s.split(sep)` splits `s` up into a list of strings, each of which appeared in `s` but were separated by `sep`. `s.split()` uses a single space as the separator, but ignores multiple spaces or spaces at the end. `s.splitlines()` is the same as `s.split('\n')` except that it does not give a final empty string if `s` ends with a newline.

```

1 >>> s = "abcd979efghijkl979mnopq979rst"
2 >>> s.split("979")
3     ['abcd', 'efghijkl', 'mnopq', 'rst']
4 >>> t = "the cat  sat on the mat "
5 >>> t.split(' ')
6     ['the', 'cat', '', '', 'sat', 'on', 'the', '', 'mat', '']
7 >>> t.split()
8     ['the', 'cat', 'sat', 'on', 'the', 'mat']

```

`s.strip()` removes initial and trailing white space from `s`. It does not change `s`, it creates a totally new string instead. `s.lstrip()` removes spaces only from the left, and `s.rstrip()` only from the right.

```

1 >>> s = "    one two three    "
2 >>> s.strip()

```

```

3     'one two  three'
4 >>> s.lstrip()
5     'one two  three   '
6 >>> s.rstrip()
7     '   one two  three'

```

`s.center(width)` extends `s` out to the width given by adding spaces evenly to the left and right. `s.ljust(width)` only adds spaces to the end, `s.rjust(width)` only adds spaces at the beginning. There is an optional parameter to specify that the string should be padded with something other than spaces. `zfill` is the same as `rjust` with that extra parameters set to `'0'`.

```

1 >>> s = "elephant"
2 >>> s.center(16)
3     '   elephant   '
4 >>> s.center(3)
5     'elephant'
6 >>> s.ljust(16)
7     'elephant     '
8 >>> s.rjust(16)
9     '   elephant'
10 >>> s.zfill(16)
11    '00000000elephant'
12 >>> s.center(16, '.')
13    '....elephant....'

```

`s.index(sub)` returns the position (number of characters to the left) of the first occurrence of the substring `sub`. `s.rindex(sub)` looks for the last occurrence. Annoyingly, when `sub` is not present, instead of giving a sensible impossible value like `-1` Python treats it as an error. Both methods may be given one or two extra parameters, they are positions that limit the portion of `s` that will be searched.

```

1 >>> "The Cat Sat On The Mat 1234.".index("at")
2     5
3 >>> "The Cat Sat On The Mat 1234.".rindex("at")
4     20
5 >>> "The Cat Sat On The Mat 1234.".index("at", 7)
6     9

```

## v. Conversions

`s.lower()` and `s.upper()` convert a string to lower or upper case. `s.capitalize()` returns the given string but if the first character was lower case it is changed to upper case. `s.lower()` is rather pointless, it converts lower case to upper case and upper case to lower case.

```

1 >>> "The Cat Sat On The Mat 1234.".upper()
2     'THE CAT SAT ON THE MAT 1234.'
3 >>> "The Cat Sat On The Mat 1234.".lower()
4     'the cat sat on the mat 1234.'
5 >>> "the cat sat on the mat 1234.".capitalize()
6     'The cat sat on the mat 1234.'

```

```

7 >>> "The Cat Sat On The Mat 1234.".swapcase()
8      'tHE cAT sAT oN tHE mAT 1234.'
```

The `is...` methods are true if the string is not empty and *every* character in it is of a certain kind:

- `s.isalnum()`, between 'A' and 'Z' or 'a' and 'z' or '0' and '9'.
- `s.isalpha()`, between 'A' and 'Z' or 'a' and 'z'.
- `s.isascii()`, is the numeric value between 0 and 127.
- `s.isdecimal()`, between '0' and '9'
- `s.isdigit()`, counts as a digit even in strange Unicode places
- `s.isidentifier()`, follows the Python rules for a variable name
- `s.islower()`, between 'a' and 'z'
- `s.isnumeric()`, pretty much the same as `s.isdigit()`
- `s.isprintable()`, visible, not whitespace
- `s.isupper()`, between 'A' and 'Z'
- `s.isspace()`, whitespace: spaces, tabs, newlines, etc.

```

1 >>> "Elephant".isalpha()
2      True
3 >>> "Big Elephant".isalnum()
4      False      # because of the space
```

`s.expandtabs(n)` replaces all the tab '\t' characters with the right number of spaces so that printing it would have the same effect. `n` is optional, default 8, and says how many spaces a tab is worth.

```

1 >>> s = "Horse\tOrdinary\tThree\tx\tthat"
2 >>> s
3      'Horse\tOrdinary\tThree\tx\tthat'
4 >>> s.expandtabs()
5      'Horse  Ordinary          Three  x          hat'
6      # the words start at positions 0, 8, 16, 20, and 28.
```

`ord(singlecharacter)` tells you the unicode position for a character. For example, `ord('A')` is 65, the first 128 unicode positions are the same as ASCII. The opposite function is `chr`: `chr(65)` is 'A'.

`s.translate(tab)` provides an efficient way for uniformly changing one set of characters into another, leaving others untouched. The table could be an ordinary dictionary (covered later, but you can see all that's needed from the example) specifying the mappings from one numeric character code to another, or you can use the `maketrans` method to create a special object that will do the same thing. `maketrans` is a static method of `str`, meaning that it does not need a particular string to work on, but it is still part of `str`, so before the dot, the name of the class, `str`, appears instead of some particular string.

```

1 >>> tab = { 98: 100, 114: 115 }
2 >>>          # 98, 100, 114, 115 are the ASCII codes for b, d, r, s respectively
3 >>> "abracadabra".translate(tab)
4      'adsacadadsa'
5 >>>
```

```

6 >>> tab = str.maketrans("abðΣzЯ", "+-XΩÆ7")
7       "cdbaЯ Σēeað".translate(tab)
8       'cd-+7 Ωēe+X'

```

See also the subsection on file-like objects in the section on files.

## vi. Character lists

As a *very* small convenience, the string module provides some strings that contain all possible characters of a given class. This is all of them.

```

1 >>> import string
2 >>> string.ascii_letters
3       'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
4 >>> string.ascii_lowercase
5       'abcdefghijklmnopqrstuvwxyz'
6 >>> string.ascii_uppercase
7       'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
8 >>> string.digits
9       '0123456789'
10 >>> string.hexdigits
11      '0123456789abcdefABCDEF'
12 >>> string.octdigits
13      '01234567'
14 >>> string.printable
15      '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
16      UVWXYZ!"#%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
17 >>> string.punctuation
18      '!"#%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
19 >>> string.whitespace
20      '\t\n\r\x0b\x0c'

```

## vii. Regular expressions

Regular expressions represent strings that are not exactly known, but should follow some known pattern, such as the pattern for a Python file name: at least one letter or digit, then a .py, then an optional 2 or 3. That pattern is represented by the regular expression “[a-zA-Z0-9]+\.\py[23]?”. `re.match` in its most basic usage will tell us whether or not a particular string matches such a pattern.

```

1 >>> progpatt = "[a-zA-Z0-9]+\.\py[23]?"
2 >>> m = re.match(progpatt, "one.py")
3 >>> m
4       <re.Match object; span=(0, 6), match='one.py'>
5 >>> m = re.match(progpatt, "one!py")
6 >>> m
7 >>> m == None
8       True

```

If you already know about regular expressions in Unix, then you almost know the syntax for regular expressions in Python. Here are most of the rules:

- . matches absolutely any character at all, but only a single one.
- x (where x has no other special meaning in regular expressions) will match the character x exactly. Hence the “py” in the example pattern.
- [rnx] matches any one of the characters r, n, or x.
- [^rnx] matches any single character except r, n, or x.
- [D-K] matches any one of the characters D, E, F, G, H, I, J, or K.  
and they may be run together, [A-Z0-9w] will match any single capital letter, or any digit, or a lower case w. Hence the “[a-zA-Z0-9]” in the example pattern.
- \\ If you want to match a character that does have a special meaning, just precede it with a \, but remember what \ characters do inside strings, you need to double it to make it represent itself. Hence the “\\.” in the example pattern.
- ^ matches the beginning of a string or line.
- \$ matches the end of a string or line.
- ? means that the previous thing is optional. b? will match the letter b if one appears there. If there is no b at this position, it doesn't match anything, but it also doesn't cause a failure to match. Hence the “[23]?” in the example pattern.
- + means that the previous thing may be matched as many times as it appears, but it must appear at least once. a+ would match a or aa or aaa or aaaa and so on.
- \* means that the previous thing may be matched as many times as it appears, including none at all. \* is equivalent to +?.
- {n} where n is a number in decimal, will match exactly n occurrences of the previous thing. [fj]{5} could match fjjffj.
- {m,n} where m < n will match any number between m and n inclusive occurrences.. [fj]{2,4} could match ff, fj, jffj, jjjj, fjjf, but not j or fjjff.
- ... Putting two (or more) regular expressions together matches a string that starts matching the first, and ends matching the last, and has no intervening characters. The py in the example will only match the exact sequence py. The whole example consists of five regular expressions, all of which must be matched in their proper sequence. The five are “[a-zA-Z0-9]+”, “\\.”, “p”, “y”, and “[23]?”.
- ... | ... Putting a vertical line between two or more regular expressions will match anything that matches any one of them. cat|dog|horse will match the exact string cat or dog or horse, but nothing else.
- (?:...) Just like parentheses in mathematics. The ... can be any complicated combination of regular expressions, this notation makes them into a single one. (?:a|b|cd){2:3} would match aa, ab, cdcd, acd, cdacd, cdcdcd, aaa, bab, etc.
- (...) Exactly the same, plus whatever part of the searched string matched the contents of the parentheses is recorded, so it can be

obtained from `match`'s return value, or referred to again later in this regular expression. If `m` contains the return value from `match`, then `m.groups()` will deliver a tuple of all the substrings that matched something in parentheses. It doesn't work very well when the parenthesised thing is repeated with `*` or `+` or `{}`.

`\\n` where `n` is a decimal number starting from 1, matches any exact repetition of the substring that matched the `n`th recorded parenthesised sequence. `([a-zA-Z])\\1\\1` would match any sequence of three identical letters: `ccc`, `HHH`, `ppp`, but not `cat` or `dog`.

`\\s` matches any whitespace character.

`\\w` matches any digit or an underline or anything that Unicode considers to be a letter.

And there are a lot of other minor options. The main methods are:

`re.search(pattern, string)`

searches for anywhere in the string that a match occurs, anything before or after the match is ignored.

`re.match(pattern, string)`

only succeeds if the match is at the very beginning of the string, anything after the match is ignored.

`re.fullmatch(pattern, string)`

only succeeds if the entire string is a match for the pattern.

Those three methods all return `None` if the search failed, or an `re.Match` object if it was successful. If `m` is an `re.Match` object, then

`m.span()`

is a tuple containing the character positions of where the successful match started and one plus where it ended.

`m.groups()`

is a tuple containing all the successful parenthesised matches.

```
1 >>> m = re.search("[a-z+][A-Z+]", "9876abcdHJMoiuwyio")
2 >>> m
3 <re.Match object; span=(4, 11), match='abcdHJM'>
4 >>> m.span()
5 (4, 11)
6 >>> m.groups()
7 ('abcd', 'HJM')
8 >>> m = re.fullmatch("[a-zA-Z0-9]*\\.py[23]?", "program.py")
9 >>> m
10 <re.Match object; span=(0, 10), match='program.py'>
11 >>> m.span()
12 (0, 10)
13 >>> m.groups()
14 ()
```

Regular expressions have a quite clever behaviour that it is worth being aware of. `+` and `*` are naturally greedy, they swallow up as many characters as they can.

The first example below shows this, the pattern `<.*>` could have stopped after just matching `<abcd>`, but it didn't, it went right to the end of the string. But regular expression matches try to succeed. If it turns out that `*` or `+` went too far and prevented a successful overall match, they can backtrack to previous possible ending points and try again. The second example shows that. We know that `<.*>` naturally swallows the whole string, but after that, the `x` part of the pattern would cause a failure because there is nothing left to match the `x`. But still the match succeeds.

```
1 >>> whole = "<abcd> x wxyz>"
2 >>> m = re.fullmatch("<.*>", whole)
3 >>> list(m.groups())
4     ['<abcd> x wxyz>']
5
6 >>> m = re.fullmatch("<.*> x (.*)", whole)
7 >>> list(m.groups())
8     ['<abcd>', 'wxyz>']
```

If the pattern came from an external source, and you want to use it to find an exact match, it is quite possible that it will contain characters that mean something in regular expressions, and that would spoil the search. `re.escape(s)` solves the problem by putting an `\` character before all of the troublesome ones.

```
1 >>> re.escape("3.14159 * f(x)")
2     '3\\.14159\\ \\*\\ f\\(x\\)'
```

`re.findall(pattern, string)`

returns a list of all the substrings that matched, regardless of whatever non-matching characters were between them. It will not notice overlapping matches.

`re.split(pattern, string)`

is the opposite, it returns a list of strings that contain the non-matching characters that separated or came before or after, the matches.

`re.finditer(pattern, string)`

returns an iterator for all the strings that `findall` would have found.

```
1 >>> pypat = "[a-zA-Z0-9]*\\.py[23]?"
2 >>> re.findall(pypat, "program.py, a.py, and qw.py3")
3     ['program.py', 'a.py', 'qw.py3']
4 >>> re.split(pypat, "program.py, a.py, and qw.py3")
5     ['', ',', ' ', ' ', 'and ', ' ', '']
6 >>> re.findall("[a-z]+[0-9]+[a-z]+", "abc654nhj8765ubv897ss")
7     ['abc654nhj', 'ubv897ss']
```

Notice that in that last example, the substring `"nhj8765ubv"` also matched the pattern, but it was involved in an overlap, so it wasn't found.

Processing regular expressions is quite complex, and if a big pattern is used over and over again, there is a way to speed things up considerably. A regular expression can be “compiled” to produce an object that is hard-wired to do searches very efficiently. This shows everything:

```

1 >>> pypat = "[a-zA-Z0-9]*\\.py[23]?"
2 >>> sobj = re.compile(pypat)
3 >>> sobj.search("$$$$$abc.py>>>>")
4     <re.Match object; span=(5, 11), match='abc.py'>

```

## 6. Basic iterables: lists and tuples

The term iterable comes up a lot in Python, and it is a very simple concept. It just means some kind of data container (or creator) that lets you run through its contents one at a time. Strings are iterables, so are lists and tuples, our current subject, and so are a lot of other things. Some of those other things have a strange form. For example, a range. `range(10, 20)` behaves like a list of the numbers 10 to 19, but if you look at it, you find it isn't.

```

1 >>> range(10, 20)[4]
2     14
3 >>> range(10, 20)
4     range(10, 20)
5 >>> list(range(10, 20))
6     [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

```

`range(10, 20)` just reports itself to be `range(10, 20)`. That is because it isn't really a list. It is a special kind of object created to be able to produce those numbers when called upon, but not to produce them until then. That sort of thing saves a lot of memory, and a lot of iterables work that way. You will nearly always be able to force them to produce a list or tuple or something else as the last thing above shows.

The simplest data structures are lists. They come in two forms, the tuple with round brackets and the list itself with square brackets. Lists can contain anything at all, or nothing at all. They can be indexed, starting from zero. A negative index is also acceptable, it counts backwards from the end.

```

1 >>> L = [ 99, "cat", math.pi, (8, 5, 2), []]
2 >>> L
3     [99, 'cat', 3.141592653589793, (8, 5, 2), []]
4 >>> L[1]
5     'cat'
6 >>> L[3][1]
7     5
8 >>> T = (99, "cat", math.pi, [8, 5, 2], ())
9 >>> type(L)
10    <class 'list'>
11 >>> type(T) == tuple
12    True
13 >>> len(L), len(T)      # a tuple is created for multiple results
14    (5, 5)
15 >>> L[-2]
16    (6, 5, 2)

```

The contents of square bracket lists can be changed, but round bracket tuples can't.



Lists are examples of *mutable* objects, tuples are *immutable* objects.

```
1 >>> L[2] = 12345
2 >>> L
3     [99, 'cat', 12345, (8, 5, 2), []]
4 >>> T[2] = 12345      # this causes an error
5 >>> T[3][1] = "dog"   # but this doesn't, T[3] is a list
6 >>> T
7     (99, 'cat', 3.141592653589793, [8, 'dog', 2], ())
```

In many cases, comma-separated things are automatically made into tuples.

```
1 >>> 8, 4, [5, 6, 7]
2     (8, 4, [5, 6, 7])
```

With lists and tuples, an empty pair of brackets produces the desired length-zero object. For lists, a pair of square brackets with a single item inside them makes the length-one list we would expect. But if you want a tuple of just one thing, an extraneous comma is required, otherwise it just appears to be a normal parenthesised expression.

```
1 >>> len((2)) # this causes an error, (2) is just a number
2 >>> len((2, ))
3     1
4 >>> len(())
5     0
```

## i. Operations

Lists and tuples may be appended using the ordinary + operator. This does not modify the lists being added, it creates a new list containing all the combined elements. + requires the types of its operands to be the same, both lists or both tuples.

```
1 >>> (1, 2, 3) + (9, 8)
2     (1, 2, 3, 9, 8)
3 >>> [1, 2, 3] + [9, 8]
4     [1, 2, 3, 9, 8]
```

They may also be multiplied by an int to get any number of repetitions

```
1 >>> (1, 2, 3) * 4
2     (1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
3 >>> [0] * 6
4     [0, 0, 0, 0, 0, 0]
```

Slices are available, just as with strings

```
1 >>> L = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
2 >>> L[3:7]
3     [1, 5, 9, 2]
4 >>> L[-4:]
```

```
5 [6, 5, 3, 5]
```

`index` and `count` are methods. They tell you where something appears or how many times it appears. `in` is an operator, it tells you whether a value appears or not. `not in` too.

```
1 >>> A = [ "ant", "bat", "cat", "dog", "cat", "ant", "cat" ]
2 >>> A.index("cat")
3     2
4 >>> A.count("cat")
5     3
6 >>> B = ( "ant", "bat", "cat", "dog", "cat", "ant", "cat" )
7 >>> B.index("cat")
8     2
9 >>> B.count("cat")
10    3
11 >>> "cat" in B
12    True
13 >>> "horse" in A
14    False
15 >>> "horse" not in A
16    True
```

Anything even slightly list-like can be converted into a list or tuple.

```
1 >>> list((1, 4, 2, 7))
2     [1, 2, 4, 7]
3 >>> tuple([1, 4, 2, 7])
4     (1, 2, 4, 7)
```

`range(oneafterlast)`, `range(first, onefterlast)` and `range(first, onefterlast, step)` produce a special object that can easily be converted to a list or tuple.

```
1 >>> range(1, 7)
2     range(1, 7)
3 >>> list(range(1, 7))
4     [1, 2, 3, 4, 5, 6]
5 >>> tuple(range(99, 33, -11))
6     (99, 88, 77, 66, 55, 44)
7 >>> list(range(5))
8     [0, 1, 2, 3, 4]
```

A few useful functions that operate on list-like things are provided. `sorted` does what the name suggests (except maybe it doesn't. “sorted” suggests a question rather than a command), and so do the others:

```
1 >>> sorted([4, 1, 7, 2])
2     [1, 2, 7, 4]
3 >>> sorted((4, 1, 7, 2))
4     [1, 2, 7, 4] # note: input was a tuple, output was a list
5 >>> max([5, 2, 9, 3, 6])
6     9
7 >>>
```

```

8     min((5, 2, 9, 3, 6))
9 >>> 2
10    sum((4, 1, 3, 2))
      10

```

There are two variants of the sorted function, both indicated by keyword parameters:

`reverse = True` means the items will be arranged in descending order instead of the usual ascending; ascending order may be made explicit with `reverse = False`

`key = f`, where `f` is any function, specifies how items are to be compared. If the two items are `a` and `b`, then normally `a < b` means `a` will appear before `b`. if `key = f` is specified, then `f(a) < f(b)` means `a` will appear before `b`.

Lists and tuples may be compared as though they were numbers or strings with the `<`, `>`, `<=`, and `>=` operators. The items the contain are compared one-by-one from left to right. As soon as a non-equals pair is encountered, the result of their comparison is returned

```

1 >>> (12, "cat") < (99, "dog")
2     True      # 12 comes before 99
3 >>> (12, "cat") > (99, "dog")
4     False
5 >>> (12, "cat") > (12, "dog")
6     False     # 12s are the same but cat comes before dog
7 >>> (12, "cat") < (12, "dog")
8     True
9 >>> [12, "cat"] < [12, "dog"]
10    True
11 >>> sorted([("cat", 63), ("dog", 12), ("ant", 17), ("cat", 8)])
12    [('ant', 17), ('cat', 8), ('cat', 63), ('dog', 12)]
13 >>> min([("cat", 63), ("dog", 12), ("ant", 17), ("cat", 8)])
14    ('ant', 17)

```

## ii. Comprehensions

A list or tuple comprehension allows a list or tuple to be created by applying some operation to a sequence of values.

```

1 >>> [x * x for x in (3, 6, 1, 2, 4)]
2     [9, 36, 1, 4, 16]
3 >>> [x + 10 for x in range(1, 7)]
4     [11, 12, 13, 14, 15, 16]
5 >>> [x * y for (x, y) in [(1, 3), (4, 6), (2, 7)]]
6     [3, 24, 14]

```

If a tuple is created this way, the result is a special object, not a tuple yet, but it can easily be converted into a tuple.

```

1 >>> (x + 10 for x in range(1, 7))

```

```

2     <generator object <genexpr> at 0x0000009C75BB3920>
3 >>> tuple((x + 10 for x in range(1, 7)))
4     (11, 12, 13, 14, 15, 16)

```

The list of values used may also be filtered

```

1 >>> [i * 10 for i in range(1, 25) if i % 3 == 2]
2     [20, 50, 80, 110, 140, 170, 200, 230]

```

Comprehensions may be nested.

```

1 >>> array = [ [ 3, 5, 8 ], [ 1, 6, 2 ], [ 9, 4, 7 ] ]
2 >>> [[x * x for x in row] for row in array]
3     [[9, 25, 64], [1, 36, 4], [81, 16, 49]]
4 >>> [[row[i] for row in array] for i in range(len(array))]
5     [[3, 1, 9], [5, 6, 4], [8, 2, 7]]      # transposed array
6 >>> [[x * y for x in range(1, 4)] for y in range(1, 4)]
7     [[1, 2, 3], [2, 4, 6], [3, 6, 9]]
8 >>> [x * 10 for x in [y * y for y in range(9)]]
9     [0, 10, 40, 90, 160, 250, 360, 490, 640]

```

### iii. Object identity

Everything in Python, with the exception of very small ints, is an object (struct, class, or whatever you want to call it). Objects are always accessed through pointers, but that is completely automatic, you don't have to do anything about it. When you assign an object to a variable, all you are doing is making that variable point to the object, no copying is done. If you assign the same object to two variables, you will not get two copies of the object. But if you type in two identical objects, Python will not notice that, it will make two separate objects.

The operators `is` and `is not` just compare pointers, they do not look at the objects being compared at all. The operator `==` does a deep comparison, the pointer values are ignored, the contents are recursively compared with `==` as far as is necessary. If for some reason you want to see the pointer value itself, the `id` function will reveal it. `A is B` is the same thing as `id(A) == id(B)`.

```

1 >>> T = (99, "cat", math.pi, [8, 5, 2], ())
2 >>> U = T
3 >>> V = (99, "cat", math.pi, [8, 5, 2], ())
4 >>> W = (99, "cat", math.pi, [8, 5.0001, 2], ())
5 >>> T is U
6     True
7 >>> T is V
8     False
9 >>> T == U
10    True
11 >>> T == V
12    True
13 >>> V == W
14    False
15 >>> id(T)

```

```

16      276001797712
17 >>> id(U)
18      276001797712
19 >>> id(V)
20      276001799952

```

Beware when trying to create multi-dimensional lists. If you want a five row, six column array of zeros, `[[0] * 6] * 5` seems to do the trick. But it doesn't. That expression only creates two lists. One, the inner list, is as expected, a list of six zeros. The other, the outer list, is just a list of five identical copies of *the pointer to* the inner list. The effect is seen when you try to change a single item, every item in the same columns also gets changed.

```

1 >>> A = [[0] * 6] * 5
2 >>> A[2][3] = 99
3 >>> A
4      [[0, 0, 0, 99, 0, 0], [0, 0, 0, 99, 0, 0], [0, 0, 0, 99, 0, 0],
5      [0, 0, 0, 99, 0, 0], [0, 0, 0, 99, 0, 0]]

```

The right way to create such an array is with a comprehension.

```
1 >>> A = [[0] * 6 for i in range(0, 5)]
```

the comprehension forces the `[0] * 6` to be re-evaluated five times, producing a new list at each iteration. Of course, you could go even further:

```
1 >>> A = [[0 for j in range(0, 6)] for i in range(0, 5)]
```

#### iv. Copying

If you have a mutable object and you want to keep it in two variables but don't want changes to one to be reflected in the other, you need to make a copy of it. It wouldn't be difficult to write a function that makes a copy of an object if you know what that object contains, but there is a standard library that has every possibility built into it. It is called `copy` and it has two useful methods. `copy(x)` makes and returns a shallow copy of `x`. Most objects have a `copy()` method that does exactly the same thing. Shallow means that a new object is created with the same shape and size, and assignments are used to transfer the contents of `x` into the new copy.

```

1 >>> import copy
2 >>> a = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
3 >>> b = copy.copy(a)
4 >>> a[1] = "hello"
5 >>> a
6      [[1, 2, 3], 'hello', [7, 8, 9]]
7 >>> b
8      [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```

That shows clearly that `a` and `b` are not the same object, even though they did start out identical. But ...

```

1 >>> a = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
2 >>> b = copy.copy(a)
3 >>> a[1][1] = "hello"
4 >>> a
5     [[1, 2, 3], [4, 'hello', 6], [7, 8, 9]]
6 >>> b
7     [[1, 2, 3], [4, 'hello', 6], [7, 8, 9]]

```

That shows exactly what is meant by a shallow copy. Only the “top level” thing was built anew, all of the contents are just copied pointers. A deep copy works recursively as far as it needs to. Every single thing anywhere in the object has a new copy of it made, nothing is shared. `copy.deepcopy` does that.

```

1 >>> a = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
2 >>> b = copy.deepcopy(a)
3 >>> a[1][1] = "hello"
4 >>> a
5     [[1, 2, 3], [4, 'hello', 6], [7, 8, 9]]
6 >>> b
7     [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```

## v. Updates on tuples

This seems to be against the rules:

```

1 >>> a = (9, 8, 7)
2 >>> a += (1, 2, 3)
3 >>> a
4     (9, 8, 7, 1, 2, 3)

```

but it isn't. No tuples were modified. `+=` is not defined for tuples, but when a programmer uses `+=` when `+=` isn't defined but `+` is, python substitutes `a += b` with `a = a + b`. So the ordinary plus operator was used, a totally new tuple was created and assigned to `a`. The process is illustrated here.

```

1 >>> a = (9, 8, 7)
2 >>> b = a
3 >>> a += (1, 2, 3)
4 >>> b
5     (9, 8, 7)

```

but

```

1 >>> a = [9, 8, 7]
2 >>> b = a
3 >>> a += [1, 2, 3]
4 >>> b
5     [9, 8, 7, 1, 2, 3]

```

## 7. Operations only applicable to lists

Being mutable, there are many operations that apply to lists but not to tuples. Indexes and slices can be assigned to. `del` can be used to remove things. `del` is used in many other contexts too, it generally just makes things go away. The `clear` method removes everything.

```
1 >>> L = [ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 ]
2 >>> L[4] = 555
3 >>> L
4     [9, 8, 7, 6, 555, 4, 3, 2, 1, 0]
5 >>> L[2:8] = [33, 44, 55]
6 >>> L
7     [9, 8, 33, 44, 55, 1, 0]
8 >>> L[1:2] = (100, 101, 102, 103, 104, 105)
9 >>> L
10    [9, 100, 101, 102, 103, 104, 105, 33, 44, 55, 1, 0]
11 >>> del L[5]
12 >>> L
13    [9, 100, 101, 102, 103, 105, 33, 44, 55, 1, 0]
14 >>> del L[2:-3]
15 >>> L
16    [9, 100, 55, 1, 0]
17 >>> L.clear()
18 >>> L
19    []
```

The `append` method adds a single item to the end. It doesn't return any value, it modifies the list in place.

```
1 >>> LL = [ 1, 2, 3, 4, 5 ]
2 >>> L.append(9)
3 >>> L
4     [1, 2, 3, 4, 5, 9]
5 >>> L.append([25, 24, 23])
6 >>> L
7     [1, 2, 3, 4, 5, 9, [25, 24, 23]]
```

The `pop()` method removes and returns the last item, it is fast,  $O(1)$ .

`pop(N)` removes and returns the  $N$ th item, it is not fast,  $O(N)$ .

`remove(X)` searches for the item  $X$ , and removes it,  $O(N)$ .

`reverse()` does what the name suggests, it reverses the list in place rather than creating a new list.

```
1 >>> L = [ 11, 22, 33, 44, 55, 66, 77, 88, 99 ]
2 >>> L.pop()
3     99
4 >>> L
5     [11, 22, 33, 44, 55, 66, 77, 88]
6 >>> L.pop(4)
7     55
8 >>> L
9     [11, 22, 33, 44, 66, 77, 88]
10 >>> L.remove(77)
11 >>> L
```

```

12     [11, 22, 33, 44, 66, 88]
13 >>> L.remove(999) # this causes an error
14 >>> L.reverse()
15 >>> L
16     [88, 66, 44, 33, 22, 11]

```

## 8. Sets

Sets are, like lists and tuples, collections of values. With a set there is no ordering, and each value can only appear once (adding an already present value to a set has no effect). Sets are created by putting their contents in curly brackets. `set` (some sort of collection) will create a set that contains every element of the collection. `set()` is used to create an empty set, empty curly brackets `{ }` mean something else.

```

1 >>> s = { 7, 2, "cat", 9, 2, 7, 1 }
2 >>> s
3     {1, 'cat', 2, 7, 9}
4 >>> set([2, 4, 6, 8])
5     {8, 2, 4, 6}
6 >>> type({7, 2, 4})
7     <class 'set'>
8 >>> type({})
9     <class 'dict'>
10 >>> type(set()) == set
11     True

```

The `add(x)` method adds a new item; it modifies the set in place, and returns nothing. `update(some sort of collection)` adds everything to a set. `remove(x)` removes `x` from the set, error if it isn't there. `discard(x)` is the same as `remove`, but no error if `x` wasn't there. `clear()` empties a set. The `pop()` method removes and returns one member; the member that is removed is usually the one that would appear first if the set were printed.

```

1 >>> s = { 7, 3, "cat", 9, 2, 7, 1 }
2 >>> s
3     {1, 'cat', 3, 7, 9}
4 >>> s.add(999)
5 >>> s
6     {1, 'cat', 3, 999, 7, 9}
7 >>> s.update([11, 22, 33])
8 >>> s
9     {1, 3, 7, 9, 11, 22, 33, 'cat', 999}
10 >>> s.remove(22)
11 >>> s.discard(7)
12 >>> s
13     {1, 3, 9, 11, 33, 'cat', 999}
14 >>> s.pop()
15     1
16 >>> s.pop()
17     3
18 >>>

```



```

19     s
20 >>> {9, 11, 33, 'cat', 999}
21 >>> s.clear()
22     s
      set()

```

The traditional set operations are available. & is for intersection, | is for union, - is for difference, > for superset, == for exact equality, <= for subset or equal, ^ selects items that are in one set but not both, in tests for membership, and not in is for non-membership.

```

1 >>> { 1, 4, 7, 3, 8, 9 } & { 2, 1, 6, 4, 5, 8 }
2     {8, 1, 4}
3 >>> { 7, 2, 4, 5 } | { 2, 8, 6, 4 }
4     {2, 4, 5, 6, 7, 8}
5 >>> { 7, 2, 4, 5 } - { 4, 1, 6, 7 }
6     {2, 5}
7 >>> { 7, 2, 4, 5 } ^ { 4, 1, 6, 7 }
8     {1, 2, 5, 6}
9 >>> { 3, 1, 6 } <= { 4, 1, 3, 7, 6, 8 }
10    True
11 >>> 6 in { 8, 4, 2, 7, 1 }
12    False

```

Sets may also be generated from comprehensions, and used in them, just like lists and tuples

```

1 >>> {i * 10 for i in range(1, 25) if i % 3 == 2 }
2     {230, 200, 170, 140, 110, 80, 50, 20}
3 >>> [x ** 2 for x in {6, 4, 9, 1, 4}]
4     [1, 16, 36, 81]

```

A frozenset is exactly the same as a set, except that it is immutable.

```

1 >>> s = { 1, 4, 7, 3, 8, 9 }
2 >>> fr = frozenset(s)
3 >>> fr
4     frozenset({1, 3, 4, 7, 8, 9})
5 >>> fz = frozenset([ 6, 2, 4, 8, 1 ])
6 >>> fr & fz
7     frozenset({8, 1, 4})
8 >>> fr.add(123) # this causes an error

```

Sets are implemented as hash tables. This means that only things that Python is willing to calculate a hash value for can be members of sets. Generally only immutable things are hashable, so lists and sets themselves can not be members of sets, but tuples, strings, and numbers can. Programmers can make hashable versions of mutable objects for themselves. The section on inheritance shows how to make hashable versions of non-hashable things.

```

1 >>> hash("cat")
2     2723113519068126242

```

```

3 >>> hash((5, 3, 88, 1))
4      8429117336997265710
5 >>> hash([5, 3, 88, 1])      # this causes an error
6 >>> s = { "cat", (5, 3, 88, 1) }
7 >>> s
8      {'cat', (5, 3, 88, 1)}
9 >>> s = { "cat", [5, 3, 88, 1] } # this causes an error

```

## 9. Statements

Statements may appear all on their own in a `.py` file, in which case they are executed as they are seen when the file is imported, or inside functions. Assignments and function calls and returns have already been seen.

Most kinds of statements are perfectly clear to anyone with any experience of programming. With those I'll start with just a quick summary of the syntax before going on to the details. You'll probably be able to skip the details completely, but they're there just in case something comes up.

The structured statements are the (almost) usual `if`, `while`, `for`, `with` and `try`, and there is also a definitely unusual `case` statement. `try` and `case` will be covered in later sections.

Simple statements include `pass` which does nothing, `yield`, `raise`, `assert`, and `with`. `pass` is often used as a place holder for something that has not been written yet, but it can also make statements a little clearer. `yield` and `raise` will be covered later. `assert` takes an expression and a string. If the expression is false, an error is caused and the string is shown. If the expression is true, nothing happens.

```

1 >>> if x == "cat":
2 ...     pass
3 ...     else:
4 ...         print("It isn't a cat!")
5 >>> assert -7 >= 0, "It's too small"
6         # produces an error message that ends with
7         AssertionError: It's too small
8 >>> assert 7 >= 0, "It's too small"
9         # does nothing

```

`return` is only allowed inside a function. If the function is to produce a value that can be used by the caller, provide that value after the word `return`. If a function is not supposed to produce a value, just say `return` on its own. The type of what's returned from a function does not have to be consistent. As soon as a `return` is reached, the function is stopped, nothing following the `return` will be executed.

```

1 >>> def f(x):
2 ...     if x >= 0:
3 ...         return
4 ...         print("x is positive")

```

```

5 ...     else:
6 ...         return x * 2
7 >>> f(6)
8 >>> f(-5)
9     -10

```

## ii. If

Syntax outline:

```

if some condition:
    what to do if it's true
elif another condition:
    what to do if that one's true
elif another condition:
    what to do if that one's true
else:
    what to do if none of them are true

```

The `if` statement causes following statements to be executed only if certain conditions are met. It always begins with the word `if`, then an expression, then a colon. If the expression is true the following statements are executed. Those following statements must have extra indentation. As soon as the extra indentation ends, we are back to normal unconditional statements. All statements that end with a colon have the same extra indentation requirement. Immediately after an `if`, there may be any number of `elif` (short for else if) statements. `elif` is also followed by an expression and a colon. An `elif` is only checked if the preceding `if` (and any prior `elif`s associated with it) had false conditions. Under those circumstances, the `elif` condition is evaluated and if it is true the following indented statements are executed. After an `if`, and the optional `elif`s, there may be a final `else`, also followed by a colon. The statements indented under the `else` are only executed if none of the `if` and the `elif`s were selected.

```

1 >>> if x == 0:
2 ...     print("it is zero")
3 ... elif x > 0:
4 ...     print("it is positive")
5 ...     totalpos += x
6 ... else:
7 ...     print("it is negative")
8 ...     totalneg += x

```

## iii. While

Syntax outline:

```

while some condition:
    what to do so long as it's true
break and continue are allowed

```

```
else:
    what to do if we stopped naturally, not for a break
```

The `while` statement is just like the same-named statement in other languages. It is followed by an expression and a colon. The expression is evaluated, and if it is true the following indented statements are executed. Then it starts again evaluating the condition again and so on. If the expression is found to be false at the start of any iteration, the loop stops and execution continues with the following `else:` statement if there is one, or with the rest of the program if there isn't. A loop can be interrupted by a `break` statement. As soon as a `break` is encountered, the loop stops. It doesn't wait until the condition is tested again. If a `break` is executed, the following `else:` part will not be executed; `else` is only for natural termination. A `continue` statement inside a loop abandons the rest of the current iteration and goes straight on to evaluating the condition again.

```
1 >>> while x > 0:
2 ...     x -= 1
3 ...     if x % 2 == 0:
4 ...         continue
5 ...     print(x)
6 ... else:
7 ...     print("all done")
8         9
9         7
10        5
11        3
12        1
13        all done
14 >>> x = 11
15 ... while x > 0:
16 ...     x -= 1
17 ...     if x % 6 == 0:
18 ...         break
19 ...     print(x)
20 ... else:
21 ...     print("all done")
22        10
23        9
24        8
25        7
```

#### iv. For

Syntax outline:

```
for some variable in some iterable:
    what to do for every value of that variable
    break and continue are allowed
else:
    what to do if we stopped naturally, not for a break
```

The `for` statement is another form of loop, this time controlled by a variable that takes on a range of different values. When the range is exhausted, the loop stops. `for` statements can also be controlled by `break` or `continue` statements which behave exactly as they do in a `while` statement. A `for` statement can also be followed by an `else`: which is again only executed if the loop ends naturally, not with a `break`.

```
1 >>> for x in (9, 3, 7, 2):
2 ...     print(x, x * x)
3       9 81
4       3 9
5       7 49
6       2 4
7 >>> for (x, y) in [(2, 7), (8, "cat"), ("dog", 3.14159)]:
8 ...     print(y, x)
9       7 2
10      cat 8
11      3.14159 dog
```

If a string is given, its individual characters are delivered.

```
1 >>> for c in "cat":
2 ...     print(c)
3       c
4       a
5       t
```

A `for` statement can also be made to run over a given range of numbers without having to type them all explicitly. This is done with a `range` expression. `range` produces a regular sequence of numbers as though they were in a list, but the whole list is never actually created, the numbers are just generated one-by-one only as needed.

`range()` with no parameters produces an unending sequence of numbers starting with zero and increasing by 1 each time. `range(a)` is the same except that it starts with `a` instead of 0. `range(a, b)` produces a sequence starting with `a`, and ending immediately before it reaches `b`. `range(a, b, c)` is the same, but `c` is added each time instead of 1.

```
1 >>> range(3, 6)
2     range(3, 6)
3 >>> list(range(3, 6))
4     [3, 4, 5]
5 >>> for i in range(3, 6):
6 ...     print(i)
7       3
8       4
9       5
10 >>> for i in range(100, 70, -7):
11 ...     print(i)
12      100
13      93
14      86
15      79
```

## v. With

A `with` statement protects programs from damage caused by errors, or exceptions as they are properly called. We'll use a typical piece of file-handling code as an example. Files are covered properly in a later section, but it's easy to get the basics. If you want to create a file, you use the `open` function to bring it into existence, then the `write` method to give it some contents, and finally the `close` method to finish off. A lot like this:

```
1 >>> f = open("newfile.txt", "w")
2 >>> for item in to_do_list:
3 ...     result = complicated_calculation(item)
4 ...     f.write(str(item) + " -> " + str(result) + "\n")
5 >>> f.close()
```

But what if something goes wrong during all those complicated calculations? If a file isn't properly closed, it will find itself in an unknown state, and it will continue to consume resources until the entire program exits, and even what we wrote to the file before the error could be lost. The `with` statement protects us:

```
1 >>> with open("newfile.txt", "w") as f:
2 ...     for item in to_do_list:
3 ...         result = complicated_calculation(item)
4 ...         f.write(str(item) + " -> " + str(result) + "\n")
```

When an object is created at the head of a `with` statement, it is absolutely guaranteed that no matter what goes wrong while the statements of the `with` are executed, that object will be properly closed down. Programmers who define objects that give access to critical resources have to define special methods that will enable a `with` statement to work. Only the designer of an object can know what "properly closed down" will mean for it. We'll see that later on in the section on exceptions, after classes have been properly covered.

If you don't need access to the object created by `with`, perhaps you just want the guaranteed close operation, you can leave out the `as` part:

```
1 >>> with Thing():
2 ...     f(x)
```

## vi. Executable function definitions

Syntax outline:

```
def function name ( comma separated parameter names ) :
    " optional documentation string "
    what to do when the function is called
    return is allowed
```

Function definitions, unlike in most other languages, are executable statements, and they are executed when they are seen. They behave like assignment statements, assigning a function value to the function's name as though it were a variable (which it is). Functions may be defined using calls to other functions that have not yet been defined, no checking is done until the function is actually called. But when a Python file is imported, any executable statements directly within it (i.e. not inside functions) can only use functions that have already been defined. Because of the executable nature of function definitions, functions may be defined conditionally:

```
1 >>> if x > 7:
2 ...     def f(x):
3 ...         return x * x
4 ... else:
5 ...     def f(x):
6 ...         return math.sqrt(x)
```

Some complex statements, including `if`, `while` and `for`, may be compressed, with a few semi-colon separated simple statements immediately following and on the same line as the colon. Simple statements do not include other `ifs`, `whiles` or `fors`.

```
1 >>> if 3 < 5: print(66); a = 99; b = "cat"
```

Other kinds of statements, `del`, `yield`, `raise`, `match`, and `nonlocal` will be covered later on, after their pre-requisites have been covered.

## 10. Documentation and help

The `dir` function takes any object as its parameter, and returns a list of the names of all of its methods and attributes.

The `help` function also takes any object as its parameter, and prints out the documentation for that object, which can be quite extensive. For example, `help(3+4)` will tell you everything there is to know about ints in general, nothing specific to the number 7; `help(string)` will tell you everything about strings. If the parameter is a function or class name, it will give you the documentation for that particular function or class.

Whenever you define a function or class or method, the normal Python coding style says that you should also provide documentation for it. This is very easy to do. If the first thing after `class xxx:` or `def fff(...):` is a string, it is taken to be the documentation for it. Typically this will be a triple-quoted string to allow the documentation to be reasonably formatted.

`help(function)` always gives the name and parameter list for that function, immediately followed by any documentation string it may have. If there is no

documentation string, you still get the name and parameters just as they appeared after the word `def`.

`help(class)` always starts with the name of the class and its constructor's parameters in the form of a function call. Then it prints the class's documentation string, if any. Then it lists all methods along with their own documentation strings. Then it lists a few things that it calls data descriptors, and finally it lists all of the attributes (variables within the class' objects) along with their initialisations. This last part is only done for variables that are defined and initialised directly under the `class ccc:`, it knows nothing about attributes that are created by the constructor or any other methods.

## 11. Assignments with patterns and pattern matching

You can extract items from a list-like object with an assignment statement that has multiple variables in the same “shape” as the object.

```
1 >>> T = (99, "cat", math.pi, [8, 5, 2], ())
2 >>> (a, b, c, d, e) = T
3 >>> print(e, d, c, b, a)
4      () [8, 5, 2] 3.141592653589793 cat 99
5 >>> (a, b, c, [d, e, f], g) = T
6 >>> print(g, f, e, d, c, b, a)
7      () 2 5 8 3.141592653589793 cat 99
8 >>> (a, b, c) = T # this causes an error
9 >>> [a, b, c, (d, e, f), g] = T # but this doesn't
10 >>> (a, b, c) = (x, y, z)
11 >>> [a, b, c] = (x, y, z)
```

In simple cases, the variables being assigned to do not need brackets:

```
1 >>> a, b, c = [2, 4, 6]
2 >>> a, b, c = (2, 4, 6)
3 >>> a, b, c = {2, 4, 6}
4 >>> a, b, c = "cat"
5 >>> print(a, b, c, sep = ", ")
6      c, a, t
```

but a set of variables may not be assigned to, sets are supposed to be unordered, so you wouldn't know which elements correspond with which others:

```
1 >>> {a, b, c} = {2, 4, 6} # this causes an error
```

If a dictionary is used in this way, only the keys are taken

```
1 >>> a, b, c = { "cat": 1, "abc": 7, "hello": 9 }
2 >>> print(a, b, c)
3      cat abc hello
```

If one of the variables is preceded by a star, it can “soak up” a large number of values, they will always appear as a list



```

1 >>> a, *b, c = (1, 2, 3, 4, 5, 6, 7)
2 >>> print(a, b, c)
3     1 [2, 3, 4, 5, 6] 7

```

If one of the variables being assigned is an iterable, preceding it with a star expands it out into its separate values

```

1 >>> a = [9, 8, 7]
2 >>> v, w, x, y, z = 99, *a, 66
3 >>> print(v, w, x, y, z, sep = ", ")
4     99, 9, 8, 7, 66

```

Multiple assignments like this are executed so that they act as though they were simultaneous. As a result, this is a valid way to swap two values:

```

1 >>> a = 1234
2 >>> b = 9876
3 >>> a, b = b, a
4 >>> print("a =", a, ", b =", b)
5     a = 9876, b = 1234

```

Strings are pulled apart for multiple assignments

```

1 >>> a, b, c = "cat"
2 >>> print(a, b, c)
3     c a t
4 >>> a, b, *c, d, e = "Hippopotamus"
5 >>> print(a, b, c, d, e)
6     H i ['p', 'p', 'o', 'p', 'o', 't', 'a', 'm'] u s

```

`match` is a statement that allows one from a number of patterns to be assigned to, the pattern chosen being the first that matches the shape of the value being assigned. The word `match` is followed by the expression to be assigned, then a colon. Indented under the `match` is a list of case statements, each case statement consists of the word `case`, followed by a pattern of variables, followed by a colon. After the case statement comes a sequence of further indented statements that will be executed if the case's pattern is the one selected.

```

1 >>> def show(x):
2 ...     match x:
3 ...         case (a, b):          # any two item list or tuple
4 ...             print("two items", a, b)
5 ...         case list([a, b, c]):    # only a three item list
6 ...             print("three-list", a, b, c)
7 ...         case [a, b, c, d] as e:  # e is the whole thing
8 ...             print("four items", a, b, c, d, e)
9 ...         case [a, b, c, d, *e]:
10 ...             print("long list", a, b, c, d, e)
11 ...         case (a, b, c):
12 ...             print("three-tuple", a, b, c)
13 ...         case [*a]:
14 ...

```

```

15 ...     print("short list", a)
16 ...     case ("one" | "two" | "three" | "four") as a:
17 ...         print("small number", a)
18 ...     case { "jenny": a } as b:         # any dictionary which
19 ...                                         # contains "jenny" as a key
20 ...         print("dictionary with jenny =", a, "all =", b)
21 ...     case _:
22 >>>         print("something else")
23     show((1, 2))
24 >>> two items 1 2
25     show([1, 2])
26 >>> two items 1 2
27     show({1, 2})
28 >>> something else
29     show([1, 2, 3])
30 >>> three-list 1 2 3
31     show((1, 2, 3))
32 >>> three-tuple 1 2 3
33     show([1])
34 >>> short list [1]
35     show([])
36 >>> short list []
37     show((1, 2, 3, 4))
38 >>> four items 1 2 3 4 [1, 2, 3, 4]
39     show(1, 2, 3, 4, 5)

40 >>> long list 1 2 3 4 [5]
41 >>> show((1, 2, 3, 4, 5, 6, 7, 8))
42 >>> long list 1 2 3 4 [5, 6, 7, 8]
43 >>> show("two")
44 >>> small number two
45 >>> show({ "cat": 21, "jenny": 99 })
46 >>> dictionary with jenny = 99 all = {'cat': 21, 'jenny': 99}
47 >>> show(False)
48 >>> something else

```

An underline can appear multiple times in a pattern, it will match with anything. The same variable may not appear more than once. If it does, the error message is totally unhelpful. Lists and tuples can match with either lists or tuples, but the type can be made explicit, as in the second case above. Between vertical lines any patterns are allowed, not just simple values. Anything that matches any one of the vertical lined patterns will match the whole case.

## 12. Formatting for strings and output

The `pprint` module provides functions that make complicated data structures readable. For example, the plain old print function produces something like this:

```

1 >>> a = ("cat", "hat", "pat")
2 >>> b = ["mat", (2, 3, 4), 6]
3 >>> c = {"a", "b", "t", a}
4 >>> d = "hello"
5 >>>

```

```

6 >>> e = [a, b, c, d]
7 >>> f = (7, 6, 5, b)
8 >>> g = [e, f]
9     print(g)
10    [[('cat', 'hat', 'pat'), ['mat', (2, 3, 4), 6], {'cat',
11      'hat', 'pat'}, 'b', 't', 'a'}, 'hello'], (7, 6, 5, ['mat',
      (2, 3, 4), 6])]

```

It is very hard to work out what we are looking at. But:

```

1 >>> pprint.pprint(g, indent = 3)
2     [ [ ('cat', 'hat', 'pat'),
3         ['mat', (2, 3, 4), 6],
4         {'cat', 'hat', 'pat'}, 'b', 't', 'a'},
5         'hello'],
6         (7, 6, 5, ['mat', (2, 3, 4), 6])]

```

Which is definitely better, but I would have liked them to go a bit further in splitting things up, especially the last line. Apart from the thing to be printed, `g`, all of `pprint`'s parameters are optional. The default value for `indent` is 1, which just isn't enough. The example used 3, which means that every time it starts a new line the indentation will be adjusted by +3 or -3.

Other optional parameters are `width`, default 80, the maximum number of characters to allow on any line, `stream` should be a file-like object that is open for writing, that's where the result will be sent instead of appearing on your monitor, `depth` is the maximum level of list within set within tuple, etc that it is allowed to go. If the depth of printing would exceed this value `pprint` just prints ... instead.

The `pformat` function is exactly the same, except that instead of printing the result, it returns a string containing exactly what would have been printed. Naturally it doesn't accept a `stream` parameter.

If you need to print a number of things with exactly the same values for the optional parameters, you can create a `PrettyPrinter` object with all the same parameters, and just use its `pprint` or `pformat` methods, each of which takes only the object to be printed.

```

1 >>> p = pprint.PrettyPrinter(indent = 3, width = 100)
2 >>> p.pprint(g)

```

Objects can become recursive. If I create a list and store it in a variable, then append that list to itself, the list will have a reference to itself as its last element. `pprint` functions notice that and replace the recursive object with a warning inside `<` and `>`. If you want to be warned about that in advance, the `isrecursive` function and method will tell you.

```

1 >>> x = [1, 2, 3]
2 >>> x.append(x)
3 >>> pprint.pprint(x)
4     [1, 2, 3, <Recursion on list with id=2621512555072>]
5 >>> pprint.isrecursive(x)

```

Finally, the `isreadable` function or method tells you whether the result from `pprint` or `pformat` would be a proper Python expression that you could type back into Idle or give to the `eval` function to reconstruct the original object. Obviously, things like recursion and the `...` caused by reaching the depth limit would make the answer be `False`, but so would most objects, as methods and functions don't have very good representations.

## i. The % operator

This is considered to be the old style, but it is still preferred by many programmers because it is so similar to `printf` in old C. Values may be inserted into strings using

formatting specifiers that say what the type of the value is, and optionally some details modifying its appearance. `%s` for a string, `%d` for an int, `%f` for a float, `%x` or `%X` for an int in hexadecimal, `%c` for a single character. Its new replacement is more in the Python style, the idea is that Python knows what type everything is, so the programmer shouldn't have to specify it in formats. Lots of variation is possible.

```
1 >>> "The square root of %d is %f" % (2, math.sqrt(2))
2     'The square root of 2 is 1.414214'
3 >>> "The square root of %d is %.3f" % (2, math.sqrt(2))
4     'The square root of 2 is 1.414'
5 >>> "%ss eat %ss" % ("cat", "dog")
6     'cats eat dogs'
7 >>> "%X is the unicode for %c" % (937, 937)
8     '3A9 is the unicode for Ω'
```

The `ord` function can be reversed with the `%` operator as above.

```
1 >>> ord('A')
2     65
3 >>> "%c" % (65, )
4     'A'
```

Note the extra comma with the 65. What comes to the right of the `%` operator is not a list of parameters, as it may seem, but just a single value which must be a “tuple”. Tuples are coming soon, they are mostly just lists of values surrounded by round brackets. If you want a tuple of one thing, it must be followed by a comma, otherwise it will appear to be just an ordinary expression in parentheses. The `%` operator also works with `bytes` objects.

If there is only one thing to print, there is no need to make it into a tuple

```
1 >>> "%X" % 108
2     6C
```

And the right operand to % can also be a dictionary (coming soon). To say which entry you want, just put its key inside parentheses just after the % sign in the format.

```
1 >>> d = {'z': 88, 'y': 'yak', 'x': 6.41}
2 >>> "%(y)s likes %(x)s" % d
3      'yak likes 6.41'
```

## ii. Details of the % operator's format specifications

A % sign always signals the beginning of a format specification, and it is always ended by one of the type letters. The type letters are:

%a	any object, made printable with the <code>ascii</code> function
%c	int ASCII code or length one string, print as single character
%d	an int to be printed in decimal
%E	a float to be printed in “scientific” notation: 3.521E+03
%e	same as %E but 3.521e+03
%F	a float, never E notation, always just digits and a decimal point
%f	exactly the same as %F, a float, just digits and decimal point
%G	chooses between %E and %F for best appearance based on size
%g	chooses between %e and %f for best appearance based on size
%i	same as %d, an int to be printed in decimal
%o	an int to be printed in octal
%r	any object, made printable with the <code>repr</code> function
%s	any object, made printable with the <code>str</code> function
%X	an int to be printed in hexadecimal, letters ABCDEF
%x	an int to be printed in hexadecimal, letters abcdef
%%	just print %, no parameter is consumed

With the string formats %a, %r, and %s, the thing being printed does not need to be a string, it can be anything understood by the `ascii`, `repr`, or `str` functions respectively.

Just before the type letter, there may be one or two numbers separated by a decimal point.

For strings, %a, %r, and %s:

%12.20s	min = 12, max = 20
%12s	min = 12, max is unlimited
%.20s	no minimum, max = 20
%s	no minimum, max is unlimited

min is the minimum width, if the string is shorter it gets padded.

max is the maximum width, if the string is longer the end is lost.

padding is usually spaces added to the left.

if a - appears before the min, e.g. %-12s, padding is added to the right.

For floats, %e, %f, and %g:

<code>%12.20f</code>	min = 12, precision = 20
<code>%12f</code>	min = 12, precision = 6
<code>%.20f</code>	no minimum, precision = 20
<code>%.0f</code>	no minimum, precision = 0
<code>%f</code>	no minimum, precision = 6

min as with strings, minimum number of characters to be printed.

precision is the exact number of digits after the decimal point.

zero precision means no decimal point is printed.

padding is usually spaces added to the left.

if a `-` appears before the min, e.g. `%-12f`, padding is added to the right.

if a `+` appears before the min, the `+` or `-` sign is shown even for positives.

if a space appears there, positive numbers get an extra space at the left.

it is permissible to have both `+` and `-` signs, order doesn't matter.

if a `0` appears before the min and there is no `-`, padding is zeros, not spaces.

For ints, `%d`, `%o`, and `%x`:

<code>%12d</code>	min = 12
<code>%d</code>	no minimum

min as with strings, minimum number of characters to be printed.

padding is usually spaces added to the left.

if a `-` appears before the min, e.g. `%-12d`, padding is added to the right.

if a `+` appears before the min, the `+` or `-` sign is shown even for positives.

if a space appears there, positive numbers get an extra space at the left.

it is permissible to have both `+` and `-` signs, order doesn't matter.

if a `0` appears before the min and there is no `-`, padding is zeros, not spaces.

Any and all of min, max, and precision can be replaced with a `*`, in which case an extra parameter will be taken, it should be an int, and it will be used where the `*` appeared, `"%*. *f" % (15, 7, math.pi)` is the same as `"%15.7f" % (math.pi, )`.

Finally, a `#` may appear immediately after the `%`. that signals an "alternate form", which depends on the format letter:

<code>%#...f</code>	forces a decimal point even if no digits would follow.
<code>%#...g</code>	forces a decimal point even if no digits would follow.
<code>%#...h</code>	forces a decimal point even if no digits would follow.
<code>%#...o</code>	adds <code>0o</code> at the beginning.
<code>%#...x</code>	adds <code>0x</code> at the beginning.
<code>%#...X</code>	adds <code>0X</code> at the beginning.

### iii. The format method

Strings also have a `.format` method that does similar things to the `%` operator, but

in a different way. When processing `s.format(a, b, c)`, curly brackets inside `s` are used instead of `%` signs to cause a substitution. Empty curly brackets are the simplest, they just cause the next parameter to be inserted in its natural format.

```
1 >>> "{} owes me ${} still".format("Joe", 4.28)
```

```
2      'Joe owes me $4.28 still'
```

A number in the curly brackets allows parameters to be taken out of order

```
1 >>> s = "{2} is last, {1} is second, {0} is first"
2      s.format(65, "Herbert Smith", 3.1415)
3      '3.1415 is last, Herbert Smith is second, 65 is first'
```

Names in the curly brackets allow a more explicit selection.

```
1 >>> "one: {fred}, two: {jim}".format(jim = 123, fred = 987)
2      'one: 987, two: 123'
```

If the value associated with the name is an iterable, the name may be followed by an int inside square brackets to select one item. If the value is a dictionary (coming very soon), the square brackets would contain a key name instead.

```
1 >>> things = [ 123, "cat", 654 ]
2 >>> "{0[1]} should be cat".format(things)
3      'cat should be cat'
4 >>> others = { "ant": 16, "bat": 98, "cat": 8 }
5 >>> "{0[bat]} should be 98".format(others)
6      '98 should be 98'
```

#### iv. Details of the format method's format specifications

After any positioning or selection, two characters may be added to indicate the conversion that should be used to turn the value into a printable string:

```
!a      means use the ascii function.
!r      means use the repr function.
!s      means use the str function.
```

After all of that there may be a colon `:` followed by a format specification. Format specifications are quite like the `%` sequences used with the `%` operator, but not the same. The complete form is:

First, an optional *fill* specifier, this may be any character that hasn't got a special meaning inside a format already. If the thing being printed needs to be padded to meet a minimum width requirement, this is the character used to do the padding. Default is a space.

Next, an *alignment* specifier. If a *fill* was given, then the *alignment* is required, otherwise it is optional. The possibilities are:

```
<      padding is added to the right, to give left justification
>      padding is added to the left, to give right justification
^      padding is added equally to left and right, centering
=      for numbers, the sign, + or -, is put at the very left, and the
       number itself at the very right, padding is added between
       them
```

Next, an optional *sign* specifier, only used for numeric values:

```
+      print the sign, + or -, regardless of the sign of the number
-      the default, only print the sign if the number is negative
```

space add a space to the left for positive numbers

Next, an optional # which means exactly the same as for % formatting

Next, an optional 0 any left padding will be with zeros, any *fill* overrides this.

Next, an optional minimum width, also the same as with %.

Next, an optional *grouping* character, which may be either , or \_ (comma or underline), only for numeric values. The number will be displayed as groups of digits separated by this character. For decimal formats the digits are grouped in threes, others in fours. 12345678 with a comma would become 12,345,678.

Next, an optional . followed by a precision or minimum width. It means the same as it does for the % operator

Finally, an optional *type* letter, not preceded by a %. The letters c, d, E, e, F, f, G, g, o, s, X, and x means exactly what they do for the % operator. To the set, b for binary has been added, and % for percentages. The % symbol is only used for numeric values, it causes them to be multiplied by 100 and followed with a %, so the number 0.65 is rendered as 65%.

```
1 >>> "it costs ${0:,}".format(12345678)
2     'it costs $12,345,678'
3 >>> "{0} in binary is {0:b}".format(116)
4     '116 in binary is 1110100'
```

There are special format letters for dates and times, they are covered in the section on dates and times.

`format_map` is the same as `format`, except that it takes a dictionary.

```
1 >>> d = { "ant": 123, "dog": "bad", "hat": "good" }
2 >>> "{hat} a b c {ant}".format_map(d)
3     'good a b c 123'
```

## v. Formatted string literals

There is yet a third string formatting method in Python. Put the letter `f` in front of a string, and it will be reprocessed, substituting things in curly brackets with values from the string's environment. curly brackets may contain whole expressions. Big `f` strings may be spread across multiple lines just like ordinary strings.

```
1 >>> x = 12
2 >>> y = 144
3 >>> f"{x} times {y} is {x * y}"
4     '12 times 144 is 1728'
5
6 >>> ( f"{x} t"
7     ...     f"imes {y}"
8     ...     f"is {x * y}" )
9     '12 times 144 is 1728'
10
11 >>>
```



```

12 ... f""{x} t
13 ...     imes {y}
14     is {x * y}""
      '12 t\n     imes 144\nis 1728'

```

For some reason, you are allowed to use quotes inside the curly brackets to prevent evaluation, and if you want curly brackets to appear in the result, you must double them.

```

1 >>> f"{'x'} is {x}"
2     'x is 12'
3 >>> f"{{{ 'x' }}} {{is}} {x}"
4     '{x} {is} 12'

```

An equals sign: `f"{abc=}"` is a shorthand for `f"abc={abc}"`

```

1 >>> x1 = 12 * 7
2 >>> f"{x1=}"
3     x1=84

```

Directly after an expression and its optional `=`, you can add either `!a`, `!r`, or `!s`. The meaning is the same as with the `format` method.

After that, you can have a colon `:` followed by some detailed formatting specifications, these specifications are the same as those used with the `format` method, so a few examples should suffice.

```

1 >>> x = 13737
2 >>> f"{x:10}"
3     '      13737'
4     f"{x:<10}"
5 >>> '13737      '
6     f"{x:10X}"
7 >>> '      35A9'
8     f"{x:,}"
9 >>> '13,737'
10    f"{math.pi:25.17f}"
11 >>> '      3.14159265358979312'
12    f"{math.pi:*>25.17f}"
13 >>> '*****3.14159265358979312'

```

There are special format letters for dates and times, they are covered in the section on dates and times.

## 13. Dictionaries

Dictionaries are like associative arrays or very simple databases. When data values are added to a dictionary, they are associated with "key" values. Later, you can access those data values just by providing their keys. Like sets, dictionaries are implemented as hash tables, so only hashable values may be used as keys. The data values can be anything though. The section on inheritance shows how to

make hashable versions of non-hashable things. Values in a dictionary are accessed with the syntax `dictionary[key]`, which produces a value, but can also be assigned to.

```
1 >>> di = { "cat": [1, 2, 3], 98: "dog", "bat": 1234 }
2 >>> di
3     {'cat': [1, 2, 3], 98: 'dog', 'bat': 1234}
4 >>> type(di)
5     <class 'dict'>
6 >>> di[98]
7     'dog'
8 >>> di["cat"]
9     [1, 2, 3]
10 >>> di["horse"]      # this causes an error
11 >>> di["bat"]
12     1234
13 >>> di["bat"] = 765
14 >>> di["bat"] += 20
15 >>> di["bat"]
16     785
```

There is a dictionary constructor that takes named parameters, converts those names into strings, and uses them as keys.

```
1 >>> di = dict(aa = 99, bb = "cat", dd = 6, cc = 123)
2 >>> di["bb"]
3     'cat'
4 >>> di
5     {'aa': 99, 'bb': 'cat', 'dd': 6, 'cc': 123}
```

The `in` operator when used on a dictionary only looks at the keys, but the `.items()` method means that isn't a problem, it delivers a special object that can be converted into a list.

```
1 >>> di = {"cat": 34, "dog": 99, 66: 77}
2 >>> "dog" in di
3     True
4 >>> 99 in di
5     False
6 >>> for x in di:
7 ...     print(x, end = " ")
8     cat dog 66
9 >>> [w * 2 for w in di]
10    ['catcat', 'dogdog', 132]
11 >>> di.items()
12    dict_items([('cat', 34), ('dog', 99), (66, 77)])
13 >>> [(b, a) for (a, b) in di.items()]
14    [(34, 'cat'), (99, 'dog'), (77, 66)]
```

The `.keys()` method delivers a list of all the keys. Actually, it produces a special object that can be converted to a list or whatever. `.values()` is the same for the values. `del d[k]` removes an item, error if it isn't present. `.pop(k)` is the same, but it returns the value associated with `k`. `.pop(k, v)` is the same as that, but if

`k` is not present, `v` is returned instead of causing an error. `.get(k, v)` is the same but does not remove anything. The `.clear()` method empties a dictionary. `k in d` and `k not in d` check to see if a key is present. The `.popitem()` method removes and returns one (key, value) tuple. `len(d)` returns the number of values in a dictionary.

```
1 >>> di
2     {'aa': 99, 'bb': 'cat', 'dd': 6, 'cc': 123}
3 >>> di.pop("bb")
4     'cat'
5 >>> di.pop("bb", -1)
6     -1
7 >>> di.pop("bb") # this causes an error
8 >>> di.get("bb", -1)
9     -1
10 >>> a = di.keys()
11 >>> a
12     dict_keys(['aa', 'dd', 'cc'])
13 >>> a[1] # this causes an error
14 >>> a = list(a)
15 >>> a[1]
16     'dd'
17 >>> list(di.values())
18     [99, 6, 123]
19 >>> "cc" in di
20     True
21 >>> di.popitem()
22     ('cc', 123)
23 >>> len(di)
24     2
```

The `di.update()` method takes another dictionary as a parameter, and adds its contents to those `di` already has.

```
1 >>> di
2     {'aa': 99, 'dd': 6}
3 >>> di.update( { "xx": 17 } )
4 >>> di
5     {'aa': 99, 'dd': 6, 'xx': 17}
```

There are special conditional and loop statements for working with dictionaries.

```
1 >>> di
2     {'aa': 99, 'dd': 6, 'cc': 123}
3 >>> if "aa" in di:
4 ...     print("it is here:", d["aa"])
5     it is here: 99
6 >>> for k in di:
7 ...     print(k)

8     aa
9     dd
10    cc
11 >>> for (k, v) in di.items():
```

```

12 ...     print(k, v)
13         aa 99
14         dd 6
15         cc 123

```

The `|` operator produces the union of two dictionaries, but the other set operators are not provided. `d1.update(d2)` is the same as `d1 = d1 | d2` but a bit more efficient.

```

1 >>> di = {"aa": 99, "cc": 123, "dd": 6 }
2 >>> e = {"zz": 76, "yy": 41}
3 >>> (di | e)["zz"]
4     76
5 >>> di.update(e)
6 >>> di
7     {'aa': 99, 'cc': 123, 'dd': 6, 'zz': 76, 'yy': 41}

```

The `del` statement as usual makes things go away

```

1 >>> di
2     {'aa': 99, 'dd': 6, 'cc': 123}
3 >>> del di["dd"]
4 >>> di
5     {'aa': 99, 'cc': 123}

```

Dictionaries may be used with the `%` formatting operator

```

1 >>> di = {"nn": 99, "ani": "cat", "xx": 123, "len": 6}
2 >>> "%(ani)s are %(len)d inches long" % di
3     'cats are 6 inches long'

```

## 14. enum types

enums are quite common in programming languages, but Python's idea of an `enum` is a bit different from the usual. An `enum` is like a very simple new type for values, where the possible values just have unique names. `bool` is like an `enum` with just two values, `True` and `False`.

I want to create a new type that represents a vague idea of size. It will have five values, `tiny`, `small`, `medium`, `big`, and `enormous`. Here is how it is defined:

```

1 >>> from enum import Enum
2 >>> size = Enum("size", [ "tiny", "small", "medium",
3 ...                       "big", "enormous" ])

```

The Python community says that `enum` values should be spelled in all capitals, but I'm not listening.

You see that we have to import something from the `enum` module to make this work. In the definition, `size` is used for two different things, and it does make sense for them to have the same name. The variable `size` is what we need to be

able to make values of this type. The string "size" is only used when Python displays a value of this type.

```
1 >>> size
2     <enum 'size'>
3 >>> x = size.medium
4 >>> x
5     <size.medium: 3>
6 >>> x.value
7     3
8 >>> x.name
9     'medium'
10 >>> size(4)
11    <size.big: 4>
12 >>> size["tiny"]
13    <size.tiny: 1>
14 >>> size.big == size.small
15    False
```

There is an alternative syntax for defining `enums`, and it allows the programmer to decide which value each name should be associated with. This has exactly the same effect as the previous definition:

```
1 >>> class size(Enum):
2 ...     tiny = 1
3 ...     small = 2
4 ...     medium = 3
5 ...     big = 4
6 ...     enormous = 5
```

The syntax is a bit strange, but will make sense much later on when we look at “inheritance” in “classes”. You need to enter a blank line at the end of the definition.

There are some varieties of `Enum` also available. The basic `Enum` used above associates names with `ints`, but does not let you use the names as numbers. An `IntEnum` does. `IntEnums` can be used along with numbers just about everywhere that an ordinary `int` could be, but the results are no longer `IntEnums`.

```
1 >>> from enum import IntEnum
2 >>> size = IntEnum("size", [ "tiny", "small", "medium",
3 ...                          "big", "enormous" ])
4 >>> size.small
5     <size.small: 2>
6 >>> size.small + 1
7     3
8 >>> type(size.small)
9     <enum 'size'>
10 >>> type(size.small + 1)
11    <class 'int'>
```

Similarly, `StrEnum` does the same for strings:

```
1 >>> from enum import StrEnum
```

```

2 >>> size = StrEnum("size", [ "tiny", "small", "medium",
3 ...                          "big", "enormous" ])
4 >>> size.small
5     <size.small: 'small'>
6 >>> size.small + size.medium
7     'smallmedium'

```

Flag is a bit like IntEnum, but the values chosen are all powers of two, so the bitwise operations |, &, ^, ~ can be used to combine them. This time the results are still members of the Flag class, although we can't use the abbreviated form that Python uses when printing them. IntFlag is a combination of IntEnum and Flag.

```

1 >>> from enum import Flag
2 >>> size = Flag("size", [ "tiny", "small", "medium",
3 ...                       "big", "enormous" ])
4 >>> size.tiny
5     <size.tiny: 1>
6 >>> size.small
7     <size.small: 2>
8 >>> size.enormous
9     <size.enormous: 16>
10 >>> x = size.small | size.big | size.tiny
11 >>> x
12     <size.tiny|small|big: 11>
13 >>> ~ size.medium
14     <size.tiny|small|big|enormous: 27>
15 >>> list(x)
16     [<size.tiny: 1>, <size.small: 2>, <size.big: 8>]
17 >>> len(x)
18     3

```

## 15. Functions

In general function definitions have the form of structured statements. First a line beginning with the word `def`, then the name of the function, then the list of parameter names enclosed in parentheses (you still need the parentheses even if there are no parameters), and finally a colon. The statements that make up the body of the function then appear indented under that heading. Calling a function is just the same as in most modern programming languages.

```

1 >>> def biggest_of_three_cubes(a, b, c):
2 ...     acubed = a * a * a
3 ...     bcubed = b * b * b
4 ...     ccubed = c * c * c
5 ...     if acubed > bcubed and acubed > ccubed:
6 ...         return acubed
7 ...     elif bcubed > ccubed:
8 ...         return bcubed
9 ...     else:
10 ...         return ccubed
11 ...
12 >>> biggest_of_three_cubes(3, 5, 2)

```

Small functions may be compressed in the same way as for statements

```
1 >>> def f(x): x *= 2; print(x)
```

type doesn't work as expected for functions. If you want to see if something is a function, define your own dummy function and compare its type to your something's type. Unfortunately, perfectly ordinary functions like `math.sqrt` are not recognised as functions. Also, every type has a `__name__` attribute which is always a string, and always different from the `__name__` of any other type.

```
1 >>> def square(x):
2 ...     return x * x
3 ...
4 >>> type(square)
5     <class 'function'>
6 >>> type(square) == function      # this causes an error
7 >>> ... NameError: name 'function' is not defined
8
9 >>> def f():
10 ...     pass
11 ...
12 >>> function = type(f)
13
14 >>> type(square) == function
15     True                          # now it works
16
17 >>> type(math.sqrt)
18     <class 'builtin_function_or_method'>
```

Also, every type has a `__name__` attribute which is always a string, and always different from the `__name__` of any other type. That gives us a uniform but slightly less efficient way of comparing types. Unfortunately the `__name__` of a type also distinguishes between ordinary functions and those whose implementation is not in Python and are considered to be built in. And ordinary methods are considered to be just functions.

```
1 >>> type(square).__name__
2     'function'
3 >>> type(math.sqrt).__name__
4     'builtin_function_or_method'
5 >>> type(int).__name__
6     'int'
7 >>> type(None).__name__
8     'NoneType'
```

Functions may take other functions as parameters and return functions as their results. Functions may be stored in variables, lists, whatever.

```
1 >>> def applytwice(f, x):
2 ...     return f(f(x))
3 >>> def square(x):
4 ...     return x * x
```

```

5 >>> applytwice(square, 2)
6     16
7 >>> fs = [ math.sin, square, math.sqrt ]
8 >>> fs[2](81)
9     9.0

```

## ii. Inner functions

Functions may define their own private functions within them. The inner function may access the outer function's variables and parameters if it declares them `nonlocal`. When an inner function gets out of an outer function (by being returned or assigned to a global variable) it retains its context.

```

1 >>> def multiplier_by_twice(x):
2 ...     x = 2 * x
3 ...     def f(a):
4 ...         nonlocal x
5 ...         return a * x
6 ...     return f
7 >>> g = multiplier_by_twice(5)
8 >>> g(7)
9     70
10 >>> multiplier_by_twice(5)(99)
11     990
12 >>> def apply_to_all(f, L):
13 ...     R = []
14 ...     for x in L:
15 ...         R.append(f(x))
16 ...     return R
17 >>> apply_to_all(multiplier_by_twice(10), [8, 3, 5, 1])
18     [160, 60, 100, 20]

```

## iii. Keyword and unknown parameters

When a function is called, instead of putting the parameters in the right order, values can be directed to specific parameters with keyword assignments.

```

1 >>> def something(one, two, three, four):
2 ...     return [one, two, three, four]
3 >>> something(three = 333, two = 222, four = 444, one = 111)
4     [111, 222, 333, 444]

```

If you still want to use positional parameters, they must come before any named ones.

```

1 >>> something(111, 222, four = 444, three = 333)
2     [111, 222, 333, 444]

```

If you want to insist that every parameter must be passed with a keyword, make a star appear to be the first parameter in the declaration.

```

1 >>> def abc(*, one, two, three):

```



```

2 ...     return [one, two, three]
3 >>> abc(three = 3, one = 1, two = 2)
4     [1, 2, 3]

```

A starred parameter insists that all subsequent parameters are passed by keyword, and it absorbs any additional positional parameters.

```

1 >>> def ffff(one, two, *three, four, five):
2 ...     return [one, two, three, four, five]
3 >>> ffff(1, 2, 3, 4, 5, 6, 7, five = 55, four = 44)
4     [1, 2, (3, 4, 5, 6, 7), 44, 55]
5 >>> ffff(1, 2, five = 55, four = 44)
6     [1, 2, (), 44, 55]

```

A special case of this allows a function to take a variable number of parameters of any type whatsoever, just make the starred parameter be the first and only one.

```

1 >>> def fff(* a):
2 ...     print(len(a), "parameters:")
3 ...     for i in range(len(a)):
4 ...         print(" ", i, ": (", type(a[i]), ") ", a[i], sep = "")
5
6 >>> fff()
7     0 parameters:
8 >>> fff(9745, "cat", [ 2, 4, 6])
9     3 parameters:
10     0: (<class 'int'>) 9745
11     1: (<class 'str'>) cat
12     2: (<class 'list'>) [2, 4, 6]

```

#### iv. Receiving parameters as dictionaries

A double-starred parameter takes all remaining parameters, which must use keyword assignment, as a dictionary. There must be no space between the two stars.

```

1 >>> def gg(one, **two):
2 ...     return [one, two]
3 >>> gg(1, a = 6, cat = 99, dog = 4)
4     [1, {'a': 6, 'cat': 99, 'dog': 4}]

```

#### v. All possibilities

A starred parameter can also be combined with a double-starred parameter to accept just about anything imaginable.

```

1 >>> def allsorts(* a, ** k):
2 ...     print("Unnamed parameters:")
3 ...     for v in a:
4 ...         print(" ", v)
5 ...     print("named parameters:")
6 ...     for (n, v) in k.items():

```

```

7 ...     print(" ", n, ": ", v, sep = "")
8
9 >>> allsorts(9876, "cat", joe = 66, dog = "bat")
10      Unnamed parameters:
11         9876
12         cat
13      named parameters:
14         joe: 66
15         dog: bat

```

but even here, all ordinary parameters must appear before any keyword parameters.

## vi. Default values

Parameters may also be given default values

```

1 >>> def qqq(one = 1, two = 2, three = 3, four = 4):
2 ...     return [one, two, three, four]
3 >>> qqq()
4     [1, 2, 3, 4]
5 >>> qqq(111, 222)
6     [111, 222, 3, 4]
7 >>> qqq(three = 333)
8     [1, 2, 333, 4]

```

## vii. Expanding lists for parameters

When a function is called, a star before a list-like parameter takes the contents of the list and passes them to the function as individual positional parameters.

```

1 >>> def yyy(a, b, c, d):
2 ...     return [d * 2, c * 2, b * 2, a * 2]
3 >>> L = [1, 10, 100, 1000]
4 >>> yyy(* L)
5     [2000, 200, 20, 2]
6 >>> yyy(*(i * i for i in range(4)))
7 >>> [18, 8, 2, 1]

```

## viii. Expanding dictionaries for parameters

a double star before a dictionary treats the dictionary's key-value pairs as keyword parameters.

```

1 >>> def zzz(one, two, three):
2 ...     return [one, two, three]
3 >>> di = { "three": 333, "one": 111, "two": 222 }
4 >>> zzz(** di)
5     [111, 222, 333]

```

## ix. Lambda expressions

Lambda expressions create functions that do not need to be named. The word `lambda` is followed by a comma-separated list of parameter names, a colon, and on the same line, a single expression. The value of a lambda expression is a function just like any other. Its result is of course the value of the expression.

```
1 >>> apply_to_all(lambda x: x * x, [8, 3, 5, 1])
2     [64, 9, 25, 1]
```

If a lambda expression is created in a function, it brings with it a memory of that function's parameters and local variables.

```
1 >>> def fff(y):
2 ...     return lambda x : x * y
3 >>> fff(11)(100)
4     1100
5 >>> z = fff(2)
6 >>> z(12)
7     24
```

Finally, there is a very strange and seemingly useless special piece of syntax. If a function only has one parameter, and it is supposed to be something list-like, you may use a comprehension without its own set of brackets to provide that parameter. It will not appear as a list inside the function, but as a special kind of object that can easily be turned into a list or a tuple or used in a `for` statement, etc.

```
1 >>> def g(x):
2 ...     return [[tuple(x)]]
3 >>> def h(x):
4 ...     for i in x:
5 ...         print(i)
6 >>> g(i * i for i in range(5))
7     [(0, 1, 4, 9, 16)]
8 >>> h(i * i for i in range(2, 6))
9     4
10    9
11    16
12    25
```

## x. Function overloading

Python doesn't really support name overloading: every function must have a different name, but there are two ways of achieving the safe effect. The first uses only Python features that we have already seen, and can handle all cases, but it can get long and untidy. The second, using a thing called a decorator, is neater but quite restricted in what it can do.

First method, standard Python features.

Determine the largest number of parameters that any overloaded version of the function in question needs. Define a function with that many parameters, and give them all default values of `None`.

Inside the function, you can tell which version is needed by counting the number of non-`None` parameters, and checking their types in a series of `ifs`.

```
1 >>> def horse(a = None, b = None, c = None, d = None):
2 ...
3 ...     if type(a) == int and b == None:
4 ...         print("one int parameter", a)
5 ...
6 ...     elif type(a) == str and type(b) == str and c == None:
7 ...         print("two string parameters", a, b)
8 ...
9 ...     elif a == None:
10 ...         print("no parameters")
11 ...
12 ...     else:
13 ...         raise TypeError("horse: given parameters don't match")
```

Naturally you would not do all the work inside that big `if`. Once you determine the correct number and types of parameters, just call on a normal, uniquely named function to do everything. That way you can even have parameters with meaningful names. We would change `horse` so that each case is more like this

```
6 ...     elif type(a) == str and type(b) == str and c == None:
7 ...         return horse_str_str(a, b)
```

and for each case, create a function like this

```
1 >>> def horse_str_str(first_name, last_name):
2 ...     whatever
```

Second method, the `singledispatch` decorator.

the `@singledispatch` decorator from `functools` gives a lot of the functionality of overloading. Decorators are covered in a later section. For now, they are just strange and ugly syntax that begins with an `@` character.

The following shows a function called `something` that has three different definitions, a base version that is used when none of the more specific versions are applicable, and two of those more specific versions. One to be used only when the parameter is a string, and only for only when it is a list.

```
1 >>> import functools
2
3 >>> @functools.singledispatch
4 ... def something(value):
5 ...     print("base version:", value)
6
7 >>> @something.register
8 ... def _(value: str):
```

```

9 ...     print("string version: \", value, "\", sep = ")
10
11 >>> @something.register
12 ...   def _(value: list):
13 ...     print("list, len = ", len(value), " : ", value, sep = ")
14
15 >>> something("hello")
16     string version: "hello"
17 >>> something([1, 2, 3, 4])
18     list, len = 4 : [1, 2, 3, 4]
19 >>> something(77)
20     base version: 77
21 >>> something(3.14)
22     base version: 3.14

```

You start by defining the base version of the function. This is the one that will be used when no other version fits. It is perfectly alright if that means never. This has to come first, because it is where you give the function its name. Just start with the line `@functools singledispatch` and define a perfectly normal function on the next line, no extra indentation.

For every one of the specific versions, you start with a similar decorator, but this time it is `@basefunctionname.register`. Of course *basefunctionname* needs to be the name you gave the first version of the function. In the function definition that follows, two things are required. You must not give the function a name, just use an underline instead, and you must specify a type for the parameter as shown above. Whenever the originally named function, *something*, is called, the registered versions are checked for one that matches the actual parameter. If one is found it is used. If none is found then the original base version is used.

This sort of overloading is quite restricted. Type discrimination only works for the first parameter, but the different versions do not all have to have the same number of parameters.

## xi. Type hints

Python doesn't allow functions to demand parameters to be of any particular type unless the programmer explicitly checks with the `type()` function as part of the function's definition.

But “type hints” are allowed. They don't mean much except when it comes to function overloading, covered in the section on decorators. A type hint is given by following a parameter name with a colon and a type name. In the function `fff` below, the hint tells readers that the parameter is expected to be an `int`, but Python won't do anything to enforce that. Some third-party software and extensions can make use of type hints.

```

1 >>> def ggg(i : int):
2 ...     print(i, type(i))
3
4 >>> ggg(45)

```

```

5      45 <class 'int'>
6 >>> ggg(3.14)
7      3.14 <class 'float'>
8 >>> ggg("hello")
9      hello <class 'str'>
10 >>> ggg( [1, 2, 3] )
11     [1, 2, 3] <class 'list'>

```

## xii. Possibly dangerous features

`eval(s)` takes a string, which should have the form of a valid Python expression. It delivers the result that that expression would produce. It has access to all variables and functions and other things that currently exist.

```

1 >>> x = 123
2 >>> y = 99
3 >>> eval("10 * x + y")
4      1329

```

`eval(s, d)` is the same except that it uses the dictionary `d` to find that value of anything that looks like a variable. Only `d` is used for such purposes, if a variable that does not appear in `d` is used, there will be an error.

```

1 >>> eval("10 * x + y", { "y": 3, "x": 7 })
2      73

```

`exec(s)` takes a string which this time should contain a Python statement or a sequence of the separated by `\n` characters. It will execute those statements.

```

1 >>> exec("x = 10\ny = 5\nx *= y\nprint(x)\nz = x + 1")
2      50
3 >>> zzz
4      51

```

`exec(s)` may also take a dictionary as a parameter. In this case, it is still not possible to look at the value of a variable that does not appear in the dictionary, but if you assign to such a variable, `exec` will just create a temporary variable of that name, so executing the statements will not affect any existing variables.

These features are considered dangerous not because they go wrong, but because they can be exploited by malicious users. If you `exec` a string that was entered by a user, they will be able to make your program do anything they want.

## 16. Operators as functions

There is a library that provides named functions that do exactly what the standard operators do.

```

1 >>> import operator
2 >>> operator.add(3, 7)

```

```

3     10
4
5 >>> def reduce(items, func, init):
6 ...     r = init
7 ...     for x in items:
8 ...         r = func(r, x)
9 ...     return r
10 >>> reduce([1, 2, 3, 4, 5, 6], operator.add, 0)
11     21
12 >>> reduce([1, 2, 3, 4, 5, 6], operator.mul, 1)
13     720

```

These are the available operator functions. The meanings of many of them are obvious from their names, some are not. Some, like `and_`, end with underlines because otherwise they would be Python reserved words.

```

operator.abs(a) is the absolute value
operator.add(a, b) is a + b
operator.and_(a, b) is bitwise a & b
operator.call(f, a, b, ...) same as f(a, b, ...)
operator.concat(a, b) same as add, but error for numbers
operator.contains(a, b) is b in a, note reversed order
operator.countOf(a, b) is a.count(b)
operator.delitem(a, b) is del a[b]
operator.eq(a, b) is a == b
operator.floordiv(a, b) is a // b
operator.ge(a, b) is a >= b
operatorgetitem(a, b) is a[b]
operator.gt(a, b) is a > b
operator.index(a, b) is a + b
operator.indexOf(a, b) is position of first b in a
operator.inv(a) is ~ a, Python's weird idea of bitwise not
operator.invert(a) exactly the same as inv
operator.is_(a, b) is a is b
operator.is_not(a, b) is a is not b
operator.le(a, b) is a <= b
operator.lshift(a, b) is a << b
operator.lt(a, b) is a < b
operator.matmul(a, b) is a @ b
operator.mod(a, b) is a % b
operator.mul(a, b) is a * b
operator.ne(a, b) is a != b
operator.neg(a) is - a
operator.not(a) is not a
operator.or_(a, b) is bitwise a | b
operator.pos(a) is + a
operator.pow(a, b) is a ** b
operator.rshift(a, b) is a >> b
operator.setitem(a, b, c) is a[b] = c

```

```
operator.sub(a, b) is a - b
operator.truediv(a, b) is a / b
operator.truth(a) is True if a would be considered true, else False
operator.xor(a, b) is a ^ b
```

There are a number of operator functions whose names begin with `i`, and at first glance seem not to work. For example, `operator.iadd(a, b)` is supposed to implement `a += b`, but if you have a variable called `a` whose value is 7, `operator.iadd(a, 5)` leaves `a` unchanged. This is because the fact is that `a += b` is not really implemented by `operator.iadd(a, b)`, but by `a = operator.iadd(a, b)`. There is an assignment too. So it seems that `operator.add` and `operator.iadd` should be exactly the same. But they shouldn't, and the reason will become clear in the section on classes. The list of update methods is:

```
operator.iadd(a, b) is a += b
operator.iand(a, b) is bitwise a &= b
operator.iconcat(a, b) is the update version of concat
operator.ifloordiv(a, b) is a //= b
operator.ilshift(a, b) is a <<= b
operator.imatmul(a, b) is a @= b
operator.imod(a, b) is a %= b
operator.imul(a, b) is a *= b
operator.ior(a, b) is a |= b
operator.ipow(a, b) is a **= b
operator.irshift(a, b) is a >>= b
operator.isub(a, b) is a -= b
operator.itruediv(a, b) is a /= b
operator.ixor(a, b) is a ^= b
```

## 17. Special operations on functions

The `functools` module provides a few useful features.

`functools.reduce` will squash any iterable down to a single value by applying a two argument function repeatedly to combine an accumulated value with each of the elements of the iterable in turn. The accumulated value starts out equal to the iterable's first item unless you provide a third parameter to override that.

```
1 >>> functools.reduce(operator.add, [1, 2, 3, 4, 5])
2     15          # note that 1+2+3+4+5 is 15
3 >>> functools.reduce(operator.mul, [1, 2, 3, 4, 5])
4     120         # note that 1*2*3*4*5 is 120
5
6 >>>                                     # this will demonstrate the order of evaluation
7 >>> def combine(a, b):
8     ...     return "comb(" + str(a) + ", " + str(b) + ")"
9 >>> functools.reduce(combine, [1, 2, 3, 4, 5])
10     'comb(comb(comb(comb(1, 2), 3), 4), 5) '
11
```



```

12 >>>
13 >>>             # and changing the initial value
14     functools.reduce(operator.add, [1, 2, 3, 4, 5], 200)
15 >>> 215
16     functools.reduce(combine, [1, 2, 3, 4, 5], 10)
17     'comb(comb(comb(comb(comb(10, 1), 2), 3), 4), 5)'
```

`functools.partial` allows a special version of a function to be created. The result behaves like the original function except that some of its parameters have already been provided, so fewer will be needed for the call. Some trivial examples will make that perfectly clear.

```

1 >>> def takefive(a, b, c, d, e):
2 ...     return [a, b, c, d, e]
3
4 >>> takefive("z", "y", "x", "w", "v")
5     ['z', 'y', 'x', 'w', 'v']
6
7 >>> takethree = functools.partial(takefive, "A", "B")
8 >>> takethree("z", "y", "x")
9     ['A', 'B', 'z', 'y', 'x']
10
11 >>> takeone = functools.partial(takethree, "M", "N")
12 >>> takeone("z")
13     ['A', 'B', 'M', 'N', 'z']
14
15 >>> takezero = functools.partial(takeone, "Q")
16 >>> takezero()
17     ['A', 'B', 'M', 'N', 'Q']
18
19 >>> quadruple = functools.partial(operator.mul, 4)
20 >>> quadruple(7)
21     28
22
23 >>> def xyz(a, b, * c, first, second, third):
24 ...     return [a, b, c, first, second, third]
25 >>> xyz(1, 2, 3, third = 4, first = 5, second = 6)
26     [1, 2, (3,), 5, 6, 4]
27
28 >>> xyz(1, 2, third = 4, first = 5, second = 6)
29     [1, 2, (), 5, 6, 4]
30
31 >>> one = functools.partial(xyz, "A", second = "sec")
32 >>> one("bbb", third = "thi", first = "fir")
33     ['A', 'bbb', (), 'fir', 'sec', 'thi']
```

Also `functools.partialmethod` does the same thing for methods of classes, and will be covered in the section on classes.

## 18. Bytes and bytearray

These are two different but very closely related types that have a lot in common with strings. Whereas strings are based on multi-byte Unicode characters, these

consist only of single byte values. When used in a string-like way, their bytes are just ASCII characters extended in various ways to use all eight bits. They are not only for string-like uses, they are also used to store generic byte-based data. bytes objects are immutable, bytearray objects are mutable.

## i. Bytes

The `bytes` type is a lot like `str`. The objects are sequences of characters that support just about all of the usual string operations, and they too are immutable. The main difference is that `bytes` objects do not use Unicode, all characters are just 8 bit bytes using ASCII codes. This makes `bytes` useful when dealing with files and especially network communications, where protocols do not usually understand Unicode. `bytes` constants are typed like strings but with the letter `b` immediately before the opening quote. They are printed the same way. There are also constructors that provide more options.

```
1 >>> x = b"Hello"
2 >>> x
3     b'hello'
4 >>> bytes([65, 66, 67, 97])      # those are the ASCII codes you'd expect
5     b'ABCa'
6 >>> bytes((65, 66, 67, 97))
7     b'ABCa'
8 >>> bytes(range(65, 75))
9     b'ABCDEFGHJIJ'
10 >>> bytes("∂€Σ←ЯÆΩ", "utf32")
11     b'\xff\xfe\x00\x00\x02"\x00\x00\xa4 ... \x00\x00'
12 >>> bytes("∂€Σ←ЯÆΩ", "utf8")
13     b'\xe2\x88\x82\xe2\x82\xa4 ... \xd0\xaf\xc3\x86\xce\xa9'
```

The string method `encode` does the same thing:

```
1 >>> "∂€Σ←ЯÆΩ".encode("utf8")
2     b'\xe2\x88\x82\xe2\x82\xa4 ... \xd0\xaf\xc3\x86\xce\xa9'
```

In the last two examples, `"utf32"` tells the constructor how to encode the Unicode characters, it means use exactly four bytes for each of them. Most of the bytes produced happened not to be proper ASCII codes (32 to 126) so were shown in their hexadecimal form. The double quote in there is from a byte that happened to be a printable ASCII code. The `...` is where I cut parts out because the line was too long. `"utf8"` is a more compact encoding that requires a bit more effort to process, and `"utf16"` is also available, along with many variations on `"CPnnn"` for what are sometimes called code pages. A `bytes` `decode` method takes things the other way. With no parameter, `.decode` assumes standard ASCII.

```
1 >>> y = b"\xD8\xE1\xB7\xEB\xA8\xE5\x20\xD2\xB7\xE1"
2 >>> z = y.decode("CP855")
3 >>> z
4     'привет мир'
5 >>> b"ABC".decode()
6     'ABC'
7 >>>
```

```

8     bytes(z, "CP855")
      b'\xd8\xe1\xb7\xeb\xa8\xe5 \xd2\xb7\xe1'

```

As usual, `str()` will convert a bytes into a string, but probably not the string you would be hoping for. The other common conversions are also there. Indexing also works, but the result is not a character but an int, its ASCII code. The `fromhex` method requires a string with exactly two characters for each byte, but it doesn't object to spaces. `hex` is the exact opposite of `fromhex`.

```

1 >>> x = b"ABCDE"
2 >>> str(x)
3     'b'ABCDE'
4 >>> list(x)
5     [65, 66, 67, 68, 69]
6 >>> tuple(x)
7     (65, 66, 67, 68, 69)
8 >>> x[2]
9     67
10 >>> bytes.fromhex("4142 437879 7A")
11     b'ABCxyz'
12 >>> b"ABCxyz".hex()
13     '41424378797a'

```

`translate` and `maketrans` work just as they do for strings.

```

1 >>> tr = bytes.maketrans(b"abcdefghijklmnopqrstuvwxy",
2 ...                       b"ABCDEFGHIJKLMN@PQRS+UVWXYZ")
3 >>> "There Are 7 Spiders On You.".translate(tr)
4     '+HERE ARE 7 SPYDERS @N Y@U.'

```

Unsurprisingly, slicing works too, but unlike [indexing] does not deliver ints.

```

1 >>> b"abcdefghijklmnopkl"[3:8]
2     b'defgh'

```

The usual string operators `+`, `*`, `in`, `==`, `>=`, `>`, `<=`, `<`, and `!=`, along with the functions `hash(x)` and `len(x)` are also available. As are the methods `capitalize`, `center`, `count`, `endswith`, `expandtabs`, `find`, `index`, `isalnum`, `isalpha`, `isascii`, `isdigit`, `islower`, `isspace`, `isupper`, `join`, `ljust`, `lower`, `lstrip`, `partition`, `removeprefix`, `removesuffix`, `replace`, `rfind`, `rindex`, `rjust`, `rpartition`, `rstrip`, `split`, `splitlines`, `startswith`, `strip`, `swapcase`, `upper`, and `zfill`.

## ii. Bytearray

A bytearray is exactly the same as a bytes, except that it is mutable. That means that assignments to indexes and `del` are both allowed, but you can only assign a numeric ASCII code. There is nothing similar to the `b"abc"` syntax for creating a bytearray, you have to use an explicit constructor.

The constructor for a bytearray may take as its parameter:

nothing at all, make an empty bytearray.  
 a bytes or another bytearray, just copy the contents.  
 an int, make a bytearray of that length, full of zeros.  
 an iterable of ints between 0 and 255, just copy those bytes.  
 a string and an encoding, e.g. bytearray("abc", encoding = "utf-8")

```

1 >>> x = bytearray(b"abcdefghijklmnopqrstuvwxy")
2 >>> x
3     bytearray(b'abcdefghijklmnopqrstuvwxy')
4 >>> x[3]
5     100
6 >>> y = bytearray(range(65, 88))
7 >>> y
8     bytearray(b'ABCDEFGHIJKLMNOPQRSTUVW')
9 >>> x[5] = '*' # this causes an error
10 >>> x[5] = 42
11 >>> x
12     bytearray(b'abcde*ghijklmnopqrstuvwxy')
```

The decode method gives the ordinary string that the object represents, assuming ASCII encoding.

```

1 >>> y = bytearray(range(65, 88))
2 >>> y
3     bytearray(b'ABCDEFGHIJKLMNOPQRSTUVW')
4 >>> y.decode()
5     'ABCDEFGHIJKLMNOPQRSTUVW'
```

The append method will enlarge a bytearray by adding one single byte to the end. The parameter must be an int value that fits in 8 bits. extend will add a whole sequence or iterator of suitable values to the end. insert(position, value) inserts the value into the bytearray so that it will be at the given position and none of the original contents are lost. del can remove a single value or a whole slice.

```

1 >>> x = bytearray(b"abcdefghijklmnopqrstuvwxy")
2 >>> x.append(65) # 65 is the ASCII code for A
3 >>> x.append(ord('Z'))
4 >>> x
5     bytearray(b'abcdefghijklmnopqrstuvwxyAZ')
6 >>> x.extend([68, 69, 70])
7 >>> x
8     bytearray(b'abcdefghijklmnopqrstuvwxyAZDEF')
9 >>> x.extend(bytearray(b"5678"))
10 >>> x
11     bytearray(b'abcdefghijklmnopqrstuvwxyAZDEF5678')
12 >>> x.insert(2, ord('-'))
13 >>> x
14     bytearray(b'ab-cdefghijklmnopqrstuvwxyAZDEF5678')
15 >>> del x[3:8]
16 >>> x
17     bytearray(b'ab-hijklmnopqrstuvwxyAZDEF5678')
```

Two update assignment operators, += and \*= are also allowed. So is append, but it can only take an integer ASCII code.

```
1 >>> x = bytearray(b"abcdefghij")
2 >>> x
3     bytearray(b'abcdefghij')
4 >>> x += bytearray(b"123")
5 >>> x
6     bytearray(b'abcdefghij123')
7 >>> x *= 2
8 >>> x
9     bytearray(b'abcdefghij123abcdefghij123')
```

The .copy() method delivers an exact copy of a bytearray. .pop() returns the ASCII code for the last character and permanently removes it. .remove(n) removes the first occurrence of ASCII code n. .replace(a, b) returns a new bytearray leaving the original unchanged. In the new one, every occurrence of a is replaced with b. a and b may be either bytearrays or bytes. .reverse() changes the bytearray by reversing the order of all of its bytes. On top of that, all of the bytes methods are also available.

```
1 >>> x = bytearray(b"abcdefghij")
2 >>> y = x.copy()
3 >>> y == x
4     True
5 >>> y is x
6     False
7 >>> x.pop()
8     106                # 106 is the ASCII code for 'j'
9 >>> x
10    bytearray(b'abcdefghi')
11 >>> x.remove(101)     # 101 is the ASCII code for 'e'
12 >>> x
13    bytearray(b'abcdcfghi')
14 >>> x.reverse()
15 >>> x
16    bytearray(b'ihgfdcba')
17 >>> x = bytearray(b"The cat sat on the mat")
18 >>> x.replace(bytearray(b"he"), bytearray(b"hat"))
19    bytearray(b'That cat sat on that mat')
20 >>> x.replace(b"at", b"AT*")
21    bytearray(b'The cAT* sAT* on the mAT*')
22 >>> x
23    bytearray(b'The cat sat on the mat')
```

### iii. ASCII encodings of non-ASCII data

When binary data needs to be stored using only readable characters, such as when a graphical image is embedded in an email, there is a commonly used scheme known as base 64 encoding. The range of upper case letters, lower case letters and digits adds up to 62 characters. Just adding '+' and '/' gives 64

characters that can be used as the 64 digits that would be necessary to write a number in base 64.

A base 64 digit is worth six binary digits, so the encoding is performed by taking the binary data six bites at a time. Every three bytes of data results in four digits of base 64. Line breaks can be inserted to prevent lines from getting too long, they are ignored on decoding. If the length of the data being encoded isn't a multiple of three bytes, then one or two '=' signs are added at the end.

Sadly there are variations in the rules for some applications. Some use '-' instead of '+', some use ',' instead of '/', some use '\_' instead of '/'. Some require '=' padding if the amount of data indicates it, some make it optional, and some forbid it. Some specify a maximum line length. There is no common core version, you have to know the circumstances that you are encoding for.

Python has a `base64` module for all of this.

```
base64.b64encode(data, altchars = None)
```

data must be a bytes or bytearray object, the returned result will be a bytes object. Newlines are not added to produce short lines. The optional altchars parameter must be a length two bytes or bytearray object that specifies the two characters to be used in place of '+' and '/'.

```
base64.b64decode(data, altchars = None, validate = False)
```

performs the obvious opposite operation. It insists that '=' padding is used when indicated, raising an exception if it is missing. If validate is False, any characters not in the set of base 64 digits is just ignored. If True they cause an exception. Even newlines cause an exception.

```
base64.urlsafe_b64encode(data) and  
base64.urlsafe_b64decode(data)
```

Are the same, but follow the slightly different rules that apply when something has to be part of a URL. '-' is used instead of '+', and '\_' instead of '/'. Note that there is only one parameter, no options. The decode option behaves as though validate were False.

There are also versions for different bases, but I'm not going to devote any more space to them. The official documentation is quite readable.

## 19. Reading and writing files

So long as you remain aware of the distinction between text files and binary files, dealing with files is quite straightforward. Nothing even needs to be imported. A text file is essentially one that can be read easily by humans without any assistance. Usually they are just a sequence of single byte character codes as defined by the ASCII standards. Things like Word's .docx files may seem at first sight to meet this condition, but they don't. Words does a lot of work to make the contents of the file human readable. A binary file is different. Think of an `int` from

a normal programming language. It will normally be by a 32-bit value, giving it a range of -2147483648 to +2147483647. That is ten digits, plus the possible minus sign, plus at least one extra character so you can tell when one number ends and the next begins, so a text file containing `ints` could need twelve bytes for each of them. But we know that 32 bits is just four bytes. Inside the computer, those `ints` take up only a third as much space. And that is all there is to a binary file. Data is stored in its internal computer format. Not only can that save a lot of space, but it also speeds up processing because all those conversions from decimal to binary are unneeded. Text files usually consist of a number of lines of characters, but there is no concept of a line in a binary file.

## i. Reading text files

There are three stages: first open the file, then read whatever you want from it, then close it. If you use Idle interactively, it is more important to remember to close files than it is in other languages. Normally when a program ends any files it had left open are automatically closed. When you are working interactively, there isn't really such a thing as a program ending. Most systems limit the number of files you can have open simultaneously, and on some systems having a file open prevents any other programs from accessing it.

To open a file, use `f = open(filename, "r")`, where `f` is any variable you choose to use. The value of `f` is not the file itself, but just enough information to be able to process it effectively and remember our current position in it. It is an error to try to open a file that doesn't exist.

The object returned by `open` is normally called a file object, but for text files its type is really `_io.TextIOWrapper`, and for binary files it is `_io.BufferedReader` or `_io.BufferedWriter`.

Then there are four methods of `f` that can extract the file's contents:

`f.read()` reads the entire file all at once, and returns the contents as a long string, with `\n` characters to represent the ends of lines.

`f.read(n)` reads `n` characters from the file and gives them to you as a string. The parameter `n` is measured in characters for text files and in bytes for binary files, Unicode means there is a difference. It doesn't care about lines. If those `n` bytes cover any line ends, the string will contain `\ns` to represent them, but it won't stop reading just because it hits the end of a line. If there are fewer than `n` bytes left in the file, you will just get a shorter string. The empty string means that you are already at the end of the file.

`f.readline()` reads from the current position to the next end of line and gives you a string with `\n` as its last character. When the file ends without a newline as its last character, the final string you get will also not have a `\n` at the end. Empty string means end of file.

`f.readlines()` reads the entire file from the current position onwards, and returns it as a list of strings. Each string, except possibly for the last, ends with a `\n` character.

There is also a special form of a for loop that has the same effect as doing `readline()` until the end is reached.

Finally, `f.close()` closes the file and frees any Python resources that were associated with keeping it open.

To illustrate, we have a file called `file.txt` containing the following three lines of text.

```
Hello,  
one two three,  
Cats eat dogs.
```

We will read the file in each of the four ways. I will use `__repr__` when printing the strings because that makes the `\n`s visible.

```
read():  
1 >>> f = open("file.txt", "r")  
2 >>> s = f.read()  
3 >>> print(s.__repr__())  
4     'Hello,\none two three,\nCats eat dogs.\n'  
5 >>> f.close()
```

```
read(n):  
1 >>> f = open("file.txt", "r")  
2 >>> while True:  
3 ...     s = f.read(10)  
4 ...     if s == "":  
5 ...         break  
6 ...     print(s.__repr__())  
7     'Hello,\none'  
8     ' two three'  
9     ',\nCats eat'  
10    ' dogs.\n'  
11 >>> f.close()
```

```
readline():  
1 >>> f = open("file.txt", "r")  
2 >>> while True:  
3 ...     s = f.readline()  
4 ...     if s == "":  
5 ...         break  
6 ...     print(s.__repr__())  
7     'Hello,\n'  
8     'one two three,\n'  
9     'Cats eat dogs.\n'  
10 >>> f.close()
```

```
readlines():  
1 >>> f = open("file.txt", "r")  
2 >>> s = f.readlines()  
3 >>> s  
4     ['Hello,\n', 'one two three,\n', 'Cats eat dogs.\n']  
5 >>> f.close()
```



```

for:
1 >>> f = open("file.txt", "r")
2 >>> for s in f:
3 ...     print(s.__repr__())
4 >>> s
5     'Hello,\n'
6     'one two three,\n'
7     'Cats eat dogs.\n'
8 >>> f.close()

```

Different computer systems have different rules for what constitutes the end of a line of text. Under windows, it is a two character sequence `\r\n` (ASCII codes 13, 10). Under Unix it is the single character `\n`, and under some older systems it was the single character `\r`. When you open a text file, Python tries to give it to you in a uniform way, so the sequence `\r\n` and a single `\r` will both be converted into a single `\n` behind the scenes. To stop this, and leave those characters untouched, just add a third parameter `newline = ""` to the `open` function call.

If a file contains unicode characters you must specify an encoding (almost certainly UTF8) when opening the file like this:

```
1 >>> fi = open("test.txt", "r", encoding = "utf8")
```

This will still work for ordinary (ASCII only) files, so it would seem that this is the safest way to open any text file. Unfortunately it isn't. If the file contains any characters whose byte values are more than 127 (commonly referred to as being in a code page) it will not work, and there is no practical way of knowing.

## ii. Writing text files

To write to a file, you use `open` and `close` in the same way as reading, except that the second parameter to `open` should be either `"w"` or `"a"` instead of `"r"`. With either of those, a new file will be created if no file with the given name exists. `"w"` means that any existing file will be overwritten, all of the original contents will be lost. `"a"` means "append", everything you write will be added to the end if the file already exists.

The only method for writing is called `write`, and it must take exactly one string as a parameter. The string is written to the file. You may use `write` as many times as needed to build up the entire file. Unlike `print`, `write` does not automatically begin a new line after each call. If you want your file to have lines of text in it, your strings must contain `\n` characters at the appropriate positions.

```

1 >>> f = open("convert.txt", "w")
2 >>> for c in range(101):
3 ...     f.write(str(c) + " centigrade is")
4 ...     f.write(" " + str(c * 9 / 5 + 32) + " fahrenheit\n")
5 >>> f.close()

```

`write` returns the number of characters written as its result. For some reason that means that that loop will produce 202 lines of annoying output. Obviously, when you're working interactively, you'd want to have the results of any commands or expressions displayed. But in a for loop? That doesn't make sense. Printing the final result of the loop, if there were one, would make perfect sense. To avoid the output, I just built a function that encloses the loop. The function was sensible and kept quiet.

Making use of a special form of the standard `print` function may be a little more convenient, as it takes as many parameters as you care to give it, and automatically applies `str` to them. This is done by giving `print` a keyword parameter called `file`. Its value should be a file object.

```
1 >>> f = open("a.txt", "w")
2 >>> print(6, "times", 9, "is", 6 * 9, file = f)
3 >>> print(list(range(10)), file = f)
4 >>> print("The end", file = f)
5 >>> f.close()
6
7 >>> f = open("a.txt", "r")
8 >>> f.readlines()
9     ['6 times 9 is 54\n',
10     '[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]\n',
11     'The end\n']
12 >>> f.close()
```

If a string contains unicode characters the file must be opened with an extra parameter specifying the encoding method to use (almost certainly UTF8) as for reading, otherwise the write or print operations will not work. This time we're in luck, it is perfectly safe to add this parameter always, so you don't have to know whether or not unicode characters will appear in advance.

```
1 >>> fo = open("test.txt", "w", encoding = "utf8")
```

### iii. Positioning and read/write files

Normally, when working on a file, you start at the beginning, work your way steadily to the end, and then stop. It doesn't have to be that way. If you are working with a very large database and know that the data you want is near the end, you don't want to have to read all the uninteresting data before that point.

`f.seek` lets you change the position in the file that the next operation will happen at. It has two parameters. The first says the position you want to go to, measured in characters. The second says what that position is relative to. 0 means from the beginning of the file, 1 means from the current position, and 2 means from the end of the file. For some reason, the only position you're allowed to ask for when measuring relative to the current position or the end of the file is 0. `seek` returns your position in the file after the requested movement. It is OK to seek past the end of the file. If you write there, the intervening character positions will be filled with blanks.

`f.seek(0, 0)` goes right back to the beginning of the file.  
`f.seek(1234, 0)` goes back or forward to the 1234th character.  
`f.seek(0, 1)` tells you your current position.  
`f.seek(0, 2)` goes right to the end of the file.

This sort of position control isn't very common with text files, but it does come up. It is a bit more common when you have a file open for reading and writing at the same time. To enable that, the second parameter to open should be "r+" or "w+". The difference is that with "r+" it is an error file the file does not exist and the file's original contents are there for you to read or overwrite. With "w+" it is OK for the file not to exist, and existing data is erased as soon as the file is opened.

```
1 >>> f = open("xxx.txt", "w+")
2 >>> f.write("The cat sat on the mat\n")
3     23
4 >>> f.write("One two three four\n")
5     19
6 >>> f.write("abcdefghijklmnopqrstuvwxy\n")
7     27
8 >>> f.write("Elephants\n")
9     10
10 >>> f.seek(22, 0)
11     22
12 >>> f.readline()
13     '\n'
14 >>> f.seek(23, 0)
15     23
16 >>> f.readline()
17     '\n'
18 >>> f.seek(24, 0)
19     24
20 >>> f.readline()
21     'One two three four\n'
22 >>> f.seek(0, 2)
23     83
24 >>> f.write("The End.\n")
25     9
26 >>> f.seek(57, 0)
27     57
28 >>> f.read(20)
29     'nopqrstuvwxyz\nElepha'
30 >>> f.close()
```

Notice that the first line written to the file was 23 characters long, but in order to re-read the second line I had to seek to position 24. This is because of the character conversions that are done at the end of a line, and I am doing this on a PC. The single `\n` that I typed at the end of each string was converted to `\r\n`, but the write method didn't include that lengthening in the result it returned. On seeking to 22, the proper end of the first line, the character seen was `\r` which was converted to `\n` when read. On seeking to 23, where the next line should begin, there was a real `\n`. It is very deceptive.

There is also an `f.tell()` method. It returns your current position in the file, measured from the beginning, without changing it.

#### iv. Binary files

To specify that a file is binary, not text, and every byte in it should be read and written exactly as it is, with no conversions, the second parameter to open should grow a letter `b`: `"rb"`, `"wb"`, `"ab"`, or `"rb+"`. Everything remains almost the same, but with these exceptions.

The `write` method must be given a `bytes` parameter instead of a string. The type `bytes` is described fully in a later section, but it is very much like a string. constants have a `b` in front of the opening quote: `b"bytes"`, and when you `[index]` to get the individual characters you get `ints`, ASCII codes, not unicode characters.

There is only one read method. When given no parameters it reads the whole file, and when given one parameter it reads that number of bytes, just as with text files. The difference is that it returns a `bytes` object instead of a string.

Positioning and closing operations remain unchanged.

#### v. Data packing

Although binary files provide a good way of efficiently and quickly storing and retrieving data, extracting ordinary data types like `ints`, `floats`, `strings` and so on from the necessary `bytes` objects can be a very awkward and error-prone process. The opposite, squeezing `ints` and `floats` and things into `bytes` objects is just as bad. Python's `struct` module goes some way towards improving this. The methods it provides exactly follow the standard rules for `structs` in C, even down to adding bytes of padding to ensure good alignment.

`struct` operations rely on two things: a format and a buffer. A format is in concept very similar to Python's `%` format strings, but in detail is very different. A buffer is usually a `bytes` or `bytearray` object that contains the data to be unpacked, or will contain the data that gets packed. Remember that when unpacking, data is being read from the buffer so `bytes` will do, but when packing, data is going into the buffer, so immutable `bytes` will not do and a `bytearray` becomes necessary. It is allowed for the buffer to be not really a `bytes` or `bytearray`, but something that behaves like one. I won't bother to mention that again in the descriptions.

Methods that do packing behave a bit like `print`, the values to be packed are provided as a sequence of individual parameters. Methods that do unpacking always return tuples, even if only one thing was unpacked. There is one exception. `iter_unpack` returns an iterator for all the extracted items instead of a tuple of them. The methods are all class methods, you never create a `struct` object.

```
pack(format, a, b, c, d, e, ...)
```

The values `a`, `b`, `c`, `d`, `e`, ... are packed according to the format to make a bytes object which is returned as the result.

```
pack_into(format, buffer, offset, a, b, c, d, e, ...)
```

The values `a`, `b`, `c`, `d`, `e`, ... are packed according to the format into the provided `bytearray` buffer. Beware: `bytearray` has `append` and `extend` methods that can enlarge a `bytearray`, but `pack_into` will not use them. You must create the `bytearray` big enough for everything you intent to insert, with the constructor `bytearray(size)`. `offset` says where to start the packing: if `offset` is 63 then `buffer[63]` is the first to receive a byte. This is so that you can conveniently build up a large buffer with multiple calls to `pack_into`.

```
unpack(format, buffer)
```

The data in the buffer is unpacked according to the given format and returned as a tuple. The size of the buffer must exactly match the size that the format would produce, even being longer than necessary is not allowed.

```
unpack_from(format, buffer, offset = 0)
```

The data in the buffer is unpacked according to the given format and returned as a tuple. The size of the buffer must be at least the size that the format would produce, being longer than necessary is allowed. `offset` says where to start the unpacking. If `offset` is 63 then the first 63 bytes will be ignored.

```
iter_unpack(format, buffer)
```

The data in `buffer` is unpacked according to `format` over and over and over again until the whole buffer has been used, creating a tuple each time. The value returned is an iterator that will produce all of those tuples. The size of `buffer` must be an exact multiple of the size dictated by `format`.

```
calcsize(format)
```

Returns the size of the `bytearray` buffer that `format` would need. This makes `unpack`'s and `iter_unpack`'s pickiness over buffer size possible to handle.

Formats are strings that describe each data value with a single character (either a letter or “?”). The single characters may be preceded by a decimal integer to provide a repetition count: `5c` is equivalent to `ccccc`. The whole format string may begin with one of five special characters that apply to the whole rest of the format.

Byte order refers to the representation of data values that are more than one byte long. For example, the number 22238 requires 16 bits (2 bytes) and is called a short int. Its hexadecimal representation is 56DE, so its two bytes are, in hexadecimal, 56 and DE. But which order are those two bytes to be stored in? Little-endian means least significant first, so the order is DE, 56. Big-endian means the opposite, most significant first giving 56, DE. In a bytes object `0x56` would appear as `v` because of the ASCII encoding, `0xDE` would appear as `\xde`. When numbers are written in decimal it makes no difference, but when numbers

are packed into bytes it makes a very big difference. All well-designed data transfer protocols make it explicit, but sometimes you will not know what the appropriate endianness is. Python adds two more possibilities: Native means whatever the C compiler that was used to create the Python you are running used. Network means what the standard network protocols use to represent IP addresses and so on. It happens to be identical to big-endian.

Size is simply the number of bytes required to represent a value, and again there are two options. Native means the same as it does for endianness. Standard means what the designers of the `struct` package decided it should be. The two are almost never going to differ.

Alignment causes extra unnecessary bytes to be inserted in order to give your data proper alignment. What it means is that a two byte value should start at a position that is a multiple of two, a four byte value should start at a position that is a multiple of four, and so on. Two examples should be enough. If your data consists of an int followed by a byte, everything is OK. Ints are four bytes long, and the starting position is always zero. If your data consists of a byte followed by an int, then storing the byte first would naturally result in the int starting at position 1, so three bytes of padding are added to bring the position up to 4. On some computers, good alignment just makes access a little faster. On some (usually very cheap) computers alignment makes no difference at all. On some, few these days, misalignment is a fatal error to a running program. `struct` allows two kinds of alignment: Native, with the same meaning as above, and None which means that no padding is ever added. If you need the end of the packed data to align properly with something, use zero repetition count with the format letter for that something.

Now with that behind us, the optional first character of a format string will be understandable. If none of these characters appears, @ is assumed.

character: @	byte order: native	size: native	alignment: native
=	native	standard	none
<	little	standard	none
>	big	standard	none
!	network	standard	none

The format letters:

letter	Python type	C type	standard size
B	int	unsigned char	1
b	int	signed char	1
c	bytes, len = 1	char	1
d	float	double	8
e	float	(none)	2
f	float	float	4
H	int	unsigned short	2
h	int	short	2
I	int	unsigned int	4
i	int	int	4

L	int	unsigned long	4
l	int	long	4
N	int	size_t	(none)
n	int	ssize_t	(none)
P	int	any pointer	(none)
p	bytes	char array	(none)
Q	int	unsigned long long	8
q	int	long long	8
s	bytes	char array	(none)
x	b"\0"	char = 0	1
?	bool	_Bool	1

N and n are only allowed with native size

P and p are only allowed with native byte order.

p is not a real C string. It is limited to 255 bytes length, the length is stored in the first byte, and there is no terminating zero byte.

s: the repetition count is not a repetition count, it is the length of the bytes object.

```

1 >>> import struct
2 >>> struct.pack("s", b"Hello")
3     b'H'
4 >>> struct.pack("5s", b"Hello")
5     b'Hello'
6 >>> struct.pack("10s", b"Hello")
7     b'Hello\x00\x00\x00\x00\x00'
8 >>> struct.pack("p", b"Hello")
9     b'\x00'
10 >>> struct.pack("2p", b"Hello")
11    b'\x01H'
12 >>> struct.pack("6p", b"Hello")
13    b'\x05Hello'
14 >>> struct.pack("10p", b"Hello")
15    b'\x05Hello\x00\x00\x00\x00'
16 >>> struct.pack("<i", 22238)
17    b'\xdeV\x00\x00'      # DE 56 00 00
18 >>> struct.pack(">i", 22238)
19    b'\x00\x00V\xde'     # 00 00 56 DE
20 >>> x = struct.pack("9h", 2, 3, 4, 8, 7, 6, 4, 7, 9)
21 >>> x
22    b'\x02\x00\x03\x00\x04\x00 ... \x00\x04\x00\x07\x00\t\x00'
23 >>> struct.unpack("9h", x)
24    (2, 3, 4, 8, 7, 6, 4, 7, 9)
25 >>> y = struct.iter_unpack("3h", x)
26 >>> list(y)
27    [(2, 3, 4), (8, 7, 6), (4, 7, 9)]
28 >>> x = struct.pack("i7p?", 7636374, b"blue", True)
29 >>> x
30    b'\x96\x85t\x00\x04blue\x00\x00\x01'
31 >>> struct.unpack("i7p?", x)
32    (7636374, b'blue', True)
33 >>> struct.unpack("hh6pBB", x)
34    (-31338, 116, b'blue', 0, 1)

```

## vi. CSV files

CSV (Comma Separated Values) files are a common way of storing very simple databases or spreadsheets. They are exactly as their name suggests. They are text files in which each line represents an individual record, and that consists of a uniform number of fields or attributes. They can be slightly complicated to process for a few reasons. There may or may not be spaces next to the commas, it doesn't have to be commas that do the separating, if a data field actually contains a comma it needs to be quoted in some way, and that brings about the question of what to do when a data field needs to contain a quote. Sometimes there is a special line at the top whose fields are not data, but the names of the fields in the data lines that follow.

Here is a small part of a very basic CSV file:

```
"atomic number", name, symbol, "atomic weight"  
1, Hydrogen, H, 1.00794  
2, Helium, He, 4.002602  
3, Lithium, Li, 6.941  
...
```

To deal with a CSV file in Python, import the `csv` module, and open the file in the normal way, specifying that ends of lines should be left alone (`newline = ""`), then use the class method `reader` or `writer` from `csv` to create a new object based on your file. These methods take an optional keyword parameter to say what the separator character is (`delimiter = ","`), the default value is a comma.

The `reader` object can be treated as an iterable (a list-like thing):

```
1 >>> import csv  
2 >>> f = open("table.csv", "r", newline = "")  
3 >>> cr = csv.reader(f)  
4 >>> for row in cr:  
5 ...     print(row)  
6     ['atomic number', 'name', 'symbol', 'atomic weight']  
7     ['1', 'Hydrogen', 'H', '1.00794']  
8     ['2', 'Helium', 'He', '4.002602']  
9     ['3', 'Lithium', 'Li', '6.941']  
10 ...  
11 >>> f.close()
```

As you can see, you get a list for every line in the file, and every piece of data is presented as a string. There is no widely accepted method for recording the proper types of each of the fields in a CSV file, the user has to be aware of what the data is, and use `int(...)`, `float(...)`, or whatever as needed.

The `writer` object has two useful methods. `writerow` is given a list, or list-like-thing, of strings and it returns the total length of the line written. `writerows` is given a list of those row lists and returns nothing.



```

1 >>> f = open("newtable.csv", "w", newline = "")
2 >>> cw = csv.writer(f)
3 >>> cw.writerow(["Henry", "Smith", "1234"])
4     18
5 >>> cw.writerow(["Jenny", "Jones", "6235"])
6     18
7 >>> cw.writerows([["Arthur", "O'Pod", "6731"],
8                   ["Sally", "X", "1984"],
9                   ["Wally", "McKay", "6935"]])
10 >>> f.close()

```

There are two variations on reader and writer, they work with dictionaries instead of lists and are called DictReader and DictWriter.

`csv.DictReader(file, names)`, like `csv.reader(file)`, builds a reading object from an already open file. The `names` parameter should be a list containing the names of the fields in each row, as strings. If the `names` parameter is `None` or left out, the field names will be taken from the first line of the file. Each row of data from the file will be returned as a dictionary.

```

1 >>> fimport csv
2 >>> f = open("table.csv", "r", newline = "")
3 >>> cdr = csv.DictReader(f)
4 >>> for row in cdr:
5 ...     print(row)
6     {'atomic number': '1', 'name': 'Hydrogen',
7      'symbol': 'H', 'atomic weight': '1.00794'}
8     {'atomic number': '2', 'name': 'Helium',
9      'symbol': 'He', 'atomic weight': '4.002602'}
10    {'atomic number': '3', 'name': 'Lithium',
11     'symbol': 'Li', 'atomic weight': '6.941'}
12
13 >>> ...
14     f.close()

```

`csv.DictWriter(file, names)` is very similar, but this time the `names` parameter is required, and the parameters you give to `writerow` must be dictionaries. The `names` parameter must be a list containing exactly the names of all the keys those dictionaries will have. It is used to determine the order that the values are written to the file. `writerows` still works too. There is also a `writeheader()` method that may be used before any calls to `writerow` or `writerows`. It writes those keys, in the order given, as the first row in the `.csv` file.

## vii. File-like objects

`io.StringIO` is a class whose objects behave as files, but only exist in memory. They allow complex strings to be built up one part at a time over a long period, or the reverse, to let you extract things from a string in the same way as you would from a file. The constructor can take nothing at all, in which case the “file” starts empty, or it can take a string which provides the initial contents.

The “file” position starts off as 0, so read operations will be able to get the whole thing. That of course means that write or print operations will overwrite the initial contents. You can avoid this by using that standard file seek method.

```
1 >>> from io import StringIO
2 >>> sio = StringIO("Elephants\none two three\nThe End\n")
3 >>> print(1, 2, 3, file = sio)
4 >>> sio.readlines()
5     ['nts\n', 'one two three\n', 'The End\n']
6 >>> sio.seek(0, 0)
7     0
8 >>> sio.readlines()
9     ['1 2 3\n', 'nts\n', 'one two three\n', 'The End\n']
10 >>> print('afterwards', file = sio)
11 >>> sio.readlines()
12     []
13 >>> sio.seek(0, 0)
14     0
15 >>> sio.readlines()
16     ['1 2 3\n', 'nts\n', 'one two three\n',
17     'The End\n', 'afterwards\n']
```

There is also an `io.BytesIO` class which behaves exactly the same way except that it is all based on `bytes` objects instead of strings.

## 20. Classes

Class definitions provide a template for creating new objects. Classes and objects usually contain variables (usually called members, but Python likes to call them attributes) and functions (which are properly called methods when they belong to a class or object). Each of those can come in two different varieties, the static or class kind, and the normal or object kind. Imagine that you have somehow invented a new kind of number, and you want to write Python programs that can work on numbers of this kind. You would define a class that says how these numbers work: what data is needed to represent them, and what operations are possible and how they are performed. From this class definition that just says general things about how everything works, you would be able to create objects or instances to be the actual numeric values of this new kind. Let's say this new kind of number is called a `newnum`. A bit of program using them might look like this:

```
1 >>> a = newnum("twenty-three")
2 >>> b = newnum("six-and-a-half")
3 >>> c = a + b
4 >>> d = c.magnitude()
5 >>> print(c)
6     twenty-nine-and-a-half
```

In that, `newnum` is the class name, and can be used as a function to create new values (objects or instances), you would have to define a method to say how that

happens. `+` is an operator on `newnums`, you would also have to define a method to say how that works. `magnitude` is just a method, which you would also have to define. `print` is the normal familiar Python `print`, you would have to define a method that tells `print` what `newnums` should look like when printed.

`+` and `magnitude` are regular methods, they can only be used if you already have a `newnum` instance for them to work on. But when `newnum` itself is used as a function, that's different. It doesn't need an existing `newnum` to work on, it just makes them. What if you wanted something that keeps track of the number of `newnums` that have been created, or the biggest `newnum` yet created in this program? Neither of those methods needs a `newnum` to work on, they are totally independent of any particular `newnum`, but are obviously something closely connected to `newnums`. And you might want to provide a value that represents  $\pi$ . That would certainly be a `newnum`, but again, it does not depend on any existing `newnum` instances, you can use it even if you have not yet created any `newnums` in your program. These last four things, the constructor `newnum(...)`, `count()`, `biggest()`, and `pi`, which don't need any existing instances to do their job are class or static methods (or attribute in the case of `pi`).

## ii. Defining a class

A class definition begins with the keyword `class`, then the name of the new class, then a colon, followed by the definitions of members and static variables, all evenly indented below the `class`. The idea of a `newnum` is too complicated for a reasonable example, so let's go with fractions instead. Every fraction is represented by a numerator and a denominator (`top` and `bot` for short) ...

```
1 >>> class fraction:
2 ...
3 ...     count = 0
4 ...
5 ...     def __init__(self, t, b):
6 ...         self.top = t
7 ...         self.bot = b
8 ...         self.simplify()
9 ...         fraction.count += 1
10 ...
11 ...     def magnitude(self):
12 ...         return abs(self.top / self.bot)
13 ...
14 ...     def simplify(self):
15 ...         div = math.gcd(self.top, self.bot)
16 ...         if div != 1:
17 ...             self.top /= div
18 ...             self.bot /= div
19 ...
20 ...     def add(self, other):
21 ...         newtop = self.top * other.bot + other.top * self.bot
22 ...         self.bot = self.bot * other.bot
23 ...         self.top = newtop
24 ...         self.simplify()
```

```

25 ...
26 >>> a = fraction(1, 4)
27 >>> b = fraction(1, 6)
28 >>> print(a.magnitude())
29     0.25
30 >>> a.add(b)
31 >>> print(fraction.count)
32     2

```

First we have the assignment `count = 0`. When an assignment appears directly inside a class definition, it creates a static attribute. It is shared by every instance ever created, and is accessible to the whole world, as demonstrated by our ability to print it right at the end. Inside methods, or in the outside world, you must refer to `count` as `fraction.count`. If you just say `count`, it will be assumed to be a new variable local only to where it appears.

Next look at the definition of `magnitude`. The best way to see how big a fraction is, and do easy comparisons, is to do the division that the fraction represents. When `a` represents the fraction  $1/4$ , we would divide `a`'s `top`, 1, by `a`'s `bot`, 4, to produce 0.25. `magnitude` is an ordinary instance method. When `a` is a fraction, you call the method to find `a`'s magnitude by writing `a.magnitude()`. Every non-static method must be given an extra first parameter, usually called `self`, but you can call it anything you want. If `a` is a fraction, then writing `a.magnitude()` really produces the function call `fraction.magnitude(a)`. The first parameter is always automatically set to the object that the method was called on, that's why it is usually called `self`. Whenever you refer to an object's attribute or method, you must connect the object to the name of the attribute or function with a dot, hence `self.top`.

Next, go down to the definition of `add`. This is not meant to represent the `+` operation, but more like the `+=` update. `a.add(b)` means that you want to change `a` by adding the value of `b` to it. Just as with the `magnitude` method the call `a.add(b)` is automatically converted to `fraction.add(a, b)`. `self` is `a` and `other` is `b`. `self.top = ...` updates `a`'s `top`. So you can see how `add` works, the normal rules for arithmetic on fractions are used to calculate `a`'s new numerator and denominator, and the result is simplified:  $2/3 + 5/12$  would naturally result in  $39/36$  which is correct but unsatisfactory, the common divisor, 3, must be factored out to produce  $13/12$ .

All that remains is `__init__`. This is how constructors are defined, they always have the name `__init__`. Whenever a class name is used to create a new object, as in `fraction(1, 4)`, first a new empty object is created and connected to the class definition so that the methods can be found when needed. Then the class' `__init__` method is called with the new empty object as its first parameter, and the remaining original parameters (1, 4) follow as the second and further parameters. All you need to do to create an ordinary instance variable or attribute is to assign something to it inside a method. If it doesn't already exist, it will be created. Of course, assigning to an attribute that does exist just updates the existing value.

Of course you would never just type a class definition into Idle like that, you'd type it into a file, maybe called `fraction.py` and import it. `fraction.py` would need `import math` at the top, and the session would really look like this:

```
1 >>> import fraction as fra
2 >>> a = fra.fraction(1, 4)
3 >>> b = fra.fraction(1, 6)
4 >>> print(a.magnitude())
5     0.25
6 >>> a.add(b)
7 >>> print(fra.fraction.count)
8     2
```

### iii. Destructors

A destructor is a method that is called when an object can never be accessed again. This happens when there is nothing left in the Python environment that contains (a reference to) the object, or when `del` is used on the object. A destructor is just a method named `__del__` that has no parameters apart from `self`.

```
1 >>> class notify_when_gone:
2 ...     def __init__(self, n):
3 ...         self.name = n
4 ...     def __del__(self):
5 ...         print(self.name, "gone")
6
7 >>> y = notify_when_gone("first")
8 >>> z = notify_when_gone("second")
9 >>> y = 66
10     first gone
11 >>> del z
12     second gone
13 >>> notify_when_gone("third")
14     <__main__.notify_when_gone object at 0x000001F0B00A59D0>
15 >>> x = notify_when_gone("fourth")
16 >>> L = [1, 2, x, 4, 5]
17 >>> x = 999
18 >>> L[2] = 3
19     fourth gone
```

The example of “third” shows that sometimes Python doesn’t notice that an object is inaccessible. The third object was never held anywhere, so as soon as it was printed it was discarded. Sometimes when I do something like this, Python notices and calls the destructor after the very next command, but as above, sometimes it doesn’t.

If you are used to programming in languages like C++, destructors are very important. They are your last chance to release any memory allocated by your object. Because Python has automatic garbage collection, this is not the case here. Any memory used for objects that are inside your object become inaccessible as

soon as the object itself does (unless something that still exists has a reference to it) so they are automatically deleted.

But Python destructors are not useless. Very often an object will have some active data that needs to be saved. If there is a chance that your own shut-down method might not be called, a destructor makes things safe. The fact that Python sometimes doesn't notice the loss of an object (which I suspect is a mistake that might be fixed soon) means that `__del__` is not up to the task. A safer solution would be to enclose everything in a `try ... except` construction (coming later) that catches every possible failure (just have an `except` for `BaseException`) and call an ordinary method for closing down at every possible exit point.

A method may be taken from an object and be used as a function independently, it retains access to the object's state:

#### iv. Special uses of methods

```
1 >>> one = fra.fraction(1, 1)
2 >>> incrementer = one.add
3 >>> thing = fraction(5, 2)
4 >>> incrementer(thing)
5     fraction(7, 2)
```

A partial method is identical to an existing method with some of its parameters already provided, an `import` from `functools` is needed first.

```
1 >>> from functools import partialmethod
2 >>> class strange:
3 ...
4 ...     def bigmethod(self, a, b, c, d, e):
5 ...         self.first = a
6 ...         self.second = b
7 ...         self.third = c
8 ...         self.fourth = d
9 ...         self.fifth = e
10 ...
11 ...     def show(self):
12 ...         print([self.first, self.second, self.third,
13 ...                 self.fourth, self.fifth])
14 ...
15 ...     mediummethod = partialmethod(bigmethod, "AAA", "BBB")
16 ...     littlemethod = partialmethod(mediummethod, "CCC", "DDD")
17 ...     emptymethod = partialmethod(littlemethod, "EEE")
18
19 >>> obj = strange()
20 >>> obj.bigmethod(1, 2, 3, 4, 5)
21 >>> obj.show()
22     [1, 2, 3, 4, 5]
23
24 >>> obj.mediummethod(11, 12, 13)
25 >>> obj.show()
26     ['AAA', 'BBB', 11, 12, 13]
27
```

```

28 >>> obj.littlemethod(1111)
29 >>> obj.show()
30     ['AAA', 'BBB', 'CCC', 'DDD', 1111]
31
32 >>> obj.emptymethod()
33 >>> obj.show()
34     ['AAA', 'BBB', 'CCC', 'DDD', 'EEE']

```

Keyword parameters can be handled exactly as with `functools.partial` from the section on special operations on functions.

## v. Constants

Python has no real concept of a constant. If somebody really wants to change something, they can probably find a way. Instead it relies on people following various conventions. If you see an attribute whose name begins with a single underline, that means that the designer of this class is telling you that you really should not change its value, correct operations depend on it only being set in certain ways. You can come close to making a constant by defining something as a property. That is described in the methods acting as attributes subsection of the section on decorators.

## vi. Useful attributes and methods

The `type` function, applied to an object, will return a `class` object. A `class` object has attributes called `__name__` and `__qualname__`, they tell you the class' name as a string. `qualname` is needed to get a unique name for a class that was defined inside another class. Every object created from a class has a `__dict__` attribute that contains all of its attributes and their values as a dictionary. Remember that `__dict__` is only for objects created from a class definition, not for Python's own objects such as `dicts` and `strs`. The `__dict__` for an ordinary object created from a class definition does not include the methods, but for a `class` object, it does. A class' `__bases__` attribute lists all the classes that it inherited from, `__mro__` shows the order they will be searched in.

A `class` object's `__subclasses__` is a method that returns a list of all the currently in-use classes that inherited from it. `class` objects also have a `__new__` method that can be used to create an object without calling the constructor. The `__new__` method must be given the class itself as a parameter. The created object will have no instance attributes, but it will still have all of its methods. Using the `fraction` example again:

```

1 >>> o = fra.fraction.__new__(fra.fraction)
2 >>> o.__dict__
3     {}
4 >>> o.simplify()    # error because o has no top or bot but simplify uses them
5 >>> o.top = 24
6 >>> o.bot = 60
7 >>> o.magnitude()
8

```

```

9 >>> 0.4
10     o.top, o.bot
11 >>> (24, 60)
12 >>> o.simplify()
13     o.top, o.bot
      (2, 5)

```

## 21. Operators and printing for objects

Now we can make things more sophisticated. We want to be able to use familiar operators like `+` or `+=` instead of having to type out long method names, and we want `print` to know how to show a fraction. To make an operator work, you just define methods with the same names as the operator... functions from the section on operators as functions, except that the names have to begin and end with `__`. For the update operators, you usually just put the letter `i` in front of the normal operator's name. Just to support addition, these two definitions would be added to the `fraction` class:

```

1 ...     def __add__(self, other):
2 ...         newtop = self.top * other.bot + other.top * self.bot
3 ...         newbot = self.bot * other.bot
4 ...         r = fraction(newtop, newbot)
5 ...         r.simplify()
6 ...         return r
7 ...
8 ...     def __iadd__(self, other):
9 ...         newtop = self.top * other.bot + other.top * self.bot
10 ...        self.bot = self.bot * other.bot
11 ...        self.top = newtop
12 ...        self.simplify()
13 ...        return self

```

Then we could say things like this:

```

1 >>> a = fra.fraction(1, 4)
2 >>> b = fra.fraction(1, 6)
3 >>> c = a + b
4 >>> a += b

```

Note that the update operators are required to return the new value, it isn't enough just to update `self`. In fact, there is no point in updating `self`, because when Python sees `a += b`, it isn't really translated to `a.__iadd__(b)`, but to `a = a.__iadd__(b)`, so updating `self` is a total waste of time. In fact we might just as well write the following, we've already done all the work for `__add__`, so why duplicate it?

```

1 ...     def __iadd__(self, other):
2 ...         return self + other

```

In fact, we could go even further. If a class has no definition for `__iadd__`, but a programmer uses `a += b` anyway, it is still OK. Python will in that case translate



it to `a = a.__add__(b)` or indeed just `a = a + b`. The `__iadd__` and so on operators are only used when you want `a += b` to mean something different from `a = a + b`, and that is usually not a good idea at all. So a perfectly reasonable way to do it is to just not bother with defining `__iadd__` at all.

## ii. Update assignments with mutability

But things are different if you consider mutability. Our fractions are mutable objects. We should probably prefer them to be immutable like other forms of number, but as things stand they are not. Things become different for update operators when different variables hold the same object. `id` and `is` can be used to detect this situation:

```
1 >>> a = fra.fraction(1, 4)
2 >>> b = fra.fraction(1, 4)
3 >>> c = a
4 >>> id(a)
5     1999823809552
6 >>> id(b)
7     1999823829648          # different from a
8 >>> id(c)
9     1999823809552          # same as a: a and b are the same object
10 >>> a is b
11     False
12 >>> a is c
13     True
14 >>> a.top = 99
15 >>> (a.top, a.bot), (b.top, b.bot), (c.top, c.bot)
16     ((99, 4), (1, 4), (99, 4))
```

With mutable objects, this is the way it is supposed to work. Changes to an object should be reflected in all variables that refer to it. But if we don't define our own `__iadd__`, then `__add__` followed by an assignment will be used, and `__add__` creates a new object, so we would see:

```
1 >>> a = fra.fraction(1, 4)
2 >>> b = fra.fraction(1, 4)
3 >>> c = a
4 >>> a += b
5 >>> id(a)
6     1999823809552
7 >>> id(b)
8     1999823829648
9 >>> id(c)
10    1999823827600
11 >>> a is c
12    False
13 >>> (a.top, a.bot), (c.top, c.bot)
14    ((1, 2), (1, 4))
```

`a` and `c` have clearly stopped sharing. But if we go back to our original version of `__iadd__`, we'll see that it doesn't create any new objects at all. It updates `self`

and returns `self`, so in `a += b`, `self` is `a`, and `a = a.__iadd__(b)` results in `a` being assigned its own `self`. Its contents have been changed, but it is the same object. It will still be shared with `c`.

You can define how (almost) any operator works for your classes just by creating a method with the same name as one of the operator-dot functions from the section on operators as functions. This only applies to those methods that actually correspond to Python operators. Those that are always used as methods, such as `a.count(b)` should just be defined as methods with exactly the same names. But you must surround their names with double underlines, as you saw in the examples. `__matmul__` is intended for matrix multiplication but can be used for anything. It gets called when you write `a @ b`. `@` is an otherwise meaningless operator with the same priority as multiplication and division.

The full list of operator method names is:

<code>+</code>	<code>__add__(self, other)</code>	<code>+=</code>	<code>__iadd__(self, other)</code>
<code>-</code>	<code>__sub__(self, other)</code>	<code>-=</code>	<code>__isub__(self, other)</code>
<code>+</code>	<code>__pos__(self)</code>	(unary)	<code>-</code> <code>__neg__(self)</code> (unary)
<code>*</code>	<code>__mul__(self, other)</code>	<code>*=</code>	<code>__imul__(self, other)</code>
<code>//</code>	<code>__floordiv__(self, other)</code>	<code>//=</code>	<code>__ifloordiv__(self, other)</code>
<code>/</code>	<code>__truediv__(self, other)</code>	<code>/=</code>	<code>__itruediv__(self, other)</code>
<code>%</code>	<code>__mod__(self, other)</code>	<code>%=</code>	<code>__imod__(self, other)</code>
<code>**</code>	<code>__pow__(self, other)</code>	<code>**=</code>	<code>__ipow__(self, other)</code>
<code>&amp;</code>	<code>__and__(self, other)</code>	<code>&amp;=</code>	<code>__iand__(self, other)</code>
<code> </code>	<code>__or__(self, other)</code>	<code> =</code>	<code>__ior__(self, other)</code>
<code>^</code>	<code>__xor__(self, other)</code>	<code>^=</code>	<code>__ixor__(self, other)</code>
<code>&lt;&lt;</code>	<code>__lshift__(self, other)</code>	<code>&lt;&lt;=</code>	<code>__ilshift__(self, other)</code>
<code>&gt;&gt;</code>	<code>__rshift__(self, other)</code>	<code>&gt;&gt;=</code>	<code>__irshift__(self, other)</code>
<code>&lt;&lt;</code>	<code>__lshift__(self, other)</code>	<code>&lt;&lt;=</code>	<code>__ilshift__(self, other)</code>
<code>&lt;</code>	<code>__lt__(self, other)</code>	<code>&gt;</code>	<code>__gt__(self, other)</code>
<code>&lt;=</code>	<code>__le__(self, other)</code>	<code>&gt;=</code>	<code>__ge__(self, other)</code>
<code>==</code>	<code>__eq__(self, other)</code>	<code>!=</code>	<code>__ne__(self, other)</code>
<code>@</code>	<code>__matmul__(self, other)</code>	<code>@=</code>	<code>__imatmul__(self, other)</code>
<code>~</code>	<code>__invert__(self)</code>		

You may have noticed that there are no special methods for defining how assignment, and, or, and not should work. There is no way to change the behaviour of an ordinary assignment statement like `a = b`. The others are absent because of Python's view of truth and falsity. Recall that in conditions, such as "if `x`:", any value of `x` that is sort-of zero like, or in some way empty, it will be taken as `True`, otherwise it will be taken as `False`. If you want some control over how your objects behave in conditions, just define a method `__bool__(self)` which simply returns `True` if you think the object should be considered true, and `False` if you don't. You can also define `__int__(self)` and `__float__(self)` to control how typecasts work.

### iii. Printing

Now for `print(...)`. There are two special methods that you will probably want to define for any new class, and they are `__str__(self)` and `__repr__(self)`.

They are both supposed to return a string that is exactly how the object should be displayed. When an object is the final result of an interactively entered computation, Python uses the `__repr__` method. When `print` is called, it uses the `__str__` method. The difference is because `print` is supposed to make nice polished friendly output, but when you're testing things interactively you are likely to be debugging, and will want to be sure you know what you're looking at. So `__repr__` returns a string that gives as much information as possible, and `__str__` returns a string that is pleasing to the eye. Perhaps we would add:

```

1 ...     def __str__(self):
2 ...         sign = ""
3 ...         t = self.top
4 ...         b = self.bot
5 ...         if b < 0:
6 ...             t = - t
7 ...             b = - b
8 ...             if t < 0:
9 ...                 t = - t
10 ...                sign = "-"
11 ...            if t == 0:
12 ...                return "0"
13 ...            if t < b:
14 ...                return sign + str(t) + "/" + str(b)
15 ...            whole = t // b
16 ...            t -= whole * b;
17 ...            if t == 0:
18 ...                return sign + str(whole)
19 ...            if sign:
20 ...                return "-" + str(whole) + " " + str(t) + "/" + str(b) + ")"
21 ...            else:
22 ...                return str(whole) + " " + str(t) + "/" + str(b)
23 ...
24 ...     def __repr__(self):
25 ...         return "fraction(" + str(self.top) + ", " + str(self.bot) + ")"

```

then

```

1 >>> fra.fraction(5, 12)
2     fraction(5, 12)
3 >>> print(fra.fraction(5, 12))
4     5/12
5 >>> fra.fraction(17, 2)
6     fraction(17, 2)
7 >>> print(fra.fraction(17, 2))
8     8 1/2
9 >>> print(fra.fraction(17, -2))
10    -(8 1/2)

```

## 22. Special methods for classes

If, somehow, your program knows the name of an attribute as a string, it can use the functions (not methods) `getattr` and `setattr` to look at or change that

attribute, and `hasattr` to check that it exists before trying to use it. Best understood when seen in action:

```
1 >>> a = fra.fraction(27, 12)
2 >>> getattr(a, "top")
3     27
4 >>> setattr(a, "top", 77)
5 >>> a
6     fraction(77, 12)
7 >>> hasattr(a, "top")
8     True
9 >>> hasattr(a, "Herbert")
10    False
```

If you want to change the way that attributes are accessed, there are three special methods. Here's an example of the first, as part of the fraction class:

```
1 ...     def __getattr__(self, name):
2 ...         if name == "sum":
3 ...             return self.top + self.bot
4 ...         elif name == "top":
5 ...             return "Access denied!"
6
7 >>> a = fra.fraction(27, 12)
8 >>> a.sum
9     39
10 >>> a.top
11     27
```

You will remember that our fractions do not have an attribute called `sum`. With this definition in place, we are pretending that there is an attribute called `sum`. Any time you look at the value of `a.xxx` or `getattr(a, "xxx")`, if `a`'s class has no attribute called `xxx`, the `__getattr__` method will be used. It only changes what happens if you try to access something that doesn't exist. Anything that does exist will just work the normal way, as you can see where the example asked for `a.top`. There is a very similar method called `__getattribute__` that is used for things that do exist as well as those that don't, but using it makes it just about impossible to do anything.

The second controls how attributes are changed. This one is used for attempts to change anything, regardless of whether it exists or not:

```
1 ...     def __setattr__(self, name, value):
2 ...         if name == "top":
3 ...             print("changing top from", self.top, "to", value)
4 ...             self.__dict__[name] = value
5
6 >>> a = fra.fraction(27, 12)
7     changing top from Access denied! to 27
8 >>> a.bot = 43
9 >>> a.top = 17
10    changing top from 27 to 17
11 >>> a
12    fraction(17, 43)
```

The possibly surprising output after the assignment `a = fra.fraction(27, 12)` is because when the constructor is called, it does the assignment `self.top = t`, that triggers the “changing top” warning. But `top` doesn't quite exist yet, it is the completion of that assignment that makes it spring into existence. So when the warning is produced, `top` is non-existent, so `__getattr__` is used. Notice that `__getattr__` doesn't cover all the possibilities. It does nothing unless the attribute is `sum` or `top`. When a function doesn't end with a `return`, Python makes it return `None`, so we will never get the normal error from trying to look at an attribute that doesn't exist.

The last line of `__setattr__`, or something like it, is required if you want to be able to do anything with these objects. Every assignment to an attribute triggers `__setattr__`, so if it is to be possible to ever assign to an attribute, even in the constructor, there must be some way to reproduce normal behaviour. This works because objects are really implemented as dictionaries. The keys are the method and attribute names, and the values are obviously their values. That dictionary is made available to us under the name `__dict__`.

The third is for when `del` is used. `del`'s job is to make things go away. If you were to say `del a.top`, then `a`'s `top` attribute would disappear. It isn't just zeroed or set to `None`, it just doesn't exist any more, `top` is removed from `a`'s `__dict__`. As well as saying `del a.top`, you could instead say `delattr(a, "top")`.

If there are attributes that must be kept at all costs, or if you just want to be warned when something is deleted, we need a `__delattr__` method, like this:

```
1 ...     def __delattr__(self, name):
2 ...         if name == "top" or name == "bot":
3 ...             printf("You can not delete", name)
4 ...         else:
5 ...             self.__dict__[name] = None
```

This makes the `top` and `bot` attributes indestructible. It also softens the behaviour on deleting any other attributes that may exist, instead of removing them from the object, it just erases their values. Normally, after saying `del a.xxx`, an access to `a.xxx` would cause an error. With this, it would not, it would produce `None` instead. But this has an unfortunate effect. If ever you `del` an attribute that doesn't exist, it springs into existence with the value `None`. That is almost certainly not what would be wanted. The last part should probably have been

```
4 ...         elif name in self.__dict__:
5 ...             self.__dict__[name] = None
```

`__getitem__` and `__setitem__` define how an object behaves when accessed with square brackets, and `__len__` tells the standard `len` function how to work. Perhaps we want to be able to pretend fractions are like arrays, with `[1]` being `top` and `[2]` being `bot`:

```
1 ...     def __getitem__(self, index):
```

```

2 ...     if index == 1:
3 ...         return self.top
4 ...     elif index == 2:
5 ...         return self.bot
6 ...     else:
7 ...         return None
8 ...
9 ...     def __setitem__(self, index, value):
10 ...         if index == 1:
11 ...             self.top = value
12 ...         elif index == 2:
13 ...             self.bot = value
14 ...
15 ...     def __len__(self):
16 ...         return 2
17
18 >>> a = fra.fraction(27, 12)
19 >>> a[1]
20     27
21 >>> a[2]
22     12
23 >>> a[1] = 99
24 >>> a
25     fraction(99, 17)
26 >>> len(a)
27     2

```

The `__call__` method is also quite useful. It lets an object behave like a function and be called.

```

1 >>> class multiplier:
2 ...
3 ...     def __init__(self, value):
4 ...         self.value = value
5 ...
6 ...     def __call__(self, other = 1):
7 ...         return self.value * other
8
9 >>> trebler = multiplier(3)
10
11 >>> trebler(6)
12     18
13 >>> trebler()
14     3
15 >>> callable(math.sqrt)
16     True
17 >>> callable([1, 2, 3])
18     False
19 >>> callable(trebler)
20     False
21 >>> callable(multiplier)
22     True

```

Here, `trebler` is an object of type `multiplier`, whose value is 3. Its `__call__` method just has one optional parameter beyond the required `self`. So when we do what looks like a function call, `trebler()`, the `__call__` method is used and the

default other of 1 is multiplied by value. If we do provide a parameter, it gets trebled. `trebler` is not a function, but it can pretend to be one.

The callable function tells you whether or not something can be used as though it were a function. `multiplier` is callable not because of its `__call__` method (it hasn't really got one, methods belong to objects, they can't be used on the class itself) but because the way we use it as a constructor, `multiplier(3)`, looks like a function call.

## ii. Automation with dataclass

Many classes that a programmer creates have obvious constructors (just provide values for all of the attributes in order as parameters) and `__repr__` methods (just show the name of the class followed by all of the attributes and their values) and so on. But Python knows nothing of obviousness, so we have to type them anyway. `dataclass` allows us to state that things should be done the obvious way, and does those things automatically. `dataclass` is an example of a decorator. We'll see how to create decorators in the next section.

```
1 >>> from dataclasses import dataclass
2
3 >>> @dataclass
4 ... class thing:
5 ...     name: str
6 ...     number: int = 0
7 ...
8 ...     def get_more(self, n):
9 ...         self.number += n
10
11 >>> a = thing("cat", 8)
12 >>> b = thing("bat")
13 >>> a
14     thing(name='cat', number=8)
15 >>> b
16     thing(name='bat', number=1)
17 >>> a == b
18     False
```

`@dataclass` must appear immediately before the class definition, and does not want extra indentation. Very un-Python-like, but there we are. With that, anything that appears to be a class variable with a type hint stops being a class variable and becomes an instance attribute instead. A type hint is a colon followed by a type name, any type will do. Type hints are not enforced in any way. For lists and such things, you should also say what it is a list of, e.g. `list[int]`. Any initial value given to the variable becomes a default value in the constructor.

The `__init__` (constructor) method is automatically added. It takes as parameters initial values for all of the known attributes, in the same order as they appeared. A `__repr__` method and an `__eq__` method are also added as shown in the example. `__eq__` just checks that all the attributes have equal values.

If you give the `dataclass` an `order` parameter (keyword only) of `True`, it will also generate `__lt__`, `__le__`, `__gt__`, and `__ge__` methods so that the comparison operators will also work. They just compare the attributes in their given order until a difference is found.

```
1 >>> @dataclass(order = True)
2 ... .. class definition as before
3 ... .. and create a and b as before
4
5 >>> a < b
6      False
```

If `frozen` is `True`, all of the attributes become read-only. Setting `eq` to `False` prevents the `__eq__` method from being generated, so the `==` operator will not work.

If `eq` and `frozen` are both `True` (`eq` is by default but `frozen` isn't) then a `__hash__` method will be generated and your objects will be able to be stored in sets and used as keys in dictionaries. Setting `unsafe_hash` to `True` forces a `__hash__` method to be created regardless of anything else.

The default value given to an attribute in a `dataclass` may be given by a call to the `field` function. All of its parameters are keyword only.

`kw_only`

If this is set to `True`, then this field's parameter in the constructor will be keyword-only. This is not the default behaviour.

`default = d`

`d` will be the default value, `x: int = field(default = 3)` is the same as `x: int = 3`.

`default_factory = f`

`f` must be something that can be used as a zero-parameter function. Every time a default value is needed for this attribute, `f` will be called to generate it. You can't have both `default` and `default_factory`.

`init default True`

If this is `False`, the constructor will not have a parameter corresponding to it.

`repr default True`

If this is `False`, this attribute will not appear in the string returned by `__repr__`.

`compare default True`

If this is `False`, this attribute will be ignored in any comparison: `==`, `!=`, `<`, `<=`, `>`, or `>=`.

`hash default None`

If this is `True` or if it is `None` and `compare` is `True` (which it is by default), this attribute will be taken into account in the `__hash__` method.

The function `fields` allows you to see all of these special attributes and their details. It may be applied to an object or the class. But the same information is



available through a few ordinary attributes, and if you only want the basics, they may be easier to deal with.

```
1 >>> a.__annotations__
2      {'name': <class 'str'>, 'number': <class 'int'>}
3 >>> a.__dataclass_params__
4      _DataclassParams(init=True, repr=True, eq=True, order=False,
5                        unsafe_hash=False, frozen=False)
6 >>> a.__dataclass_fields__
7      {'name': Field(name='name', type=<class 'str'>, default=<da
8                        ... kw_only=False, _field_type=_FIELD),
9        'number': Field(name='number', type=<class 'int'>, default
10                       ... kw_only=False, _field_type=_FIELD)}
```

Sometimes it is important that a constructor should receive some information to aid in the construction of an object, but there is no future need for that information so it would be wasteful for it to be stored as an attribute of the object. `InitVar` is for that. Instead of just saying the type of a variable you put the type inside square brackets after the word `InitVar`:

```
1 >>> capitalise_name: InitVar[bool] = False
```

`capitalise_name` will be a parameter to `__init__` with a default value of `False`, but the object created will not have a `capitalise_name` attribute.

## 23. Decorators

Static attributes and class attributes are exactly the same thing, but static methods and class methods are supposedly different. A class method is as described at the beginning of this section. Class methods have access to all of the class' class or static members or methods. Static methods don't. Except that nothing in a class is private, anything can access everything just by using the name of the class, so there really isn't much of a distinction.

### i. Class methods

To make a class method, put the symbol `@classmethod` on a line on its own, immediately before the `def` for the method. Yes, that is ugly syntax. A class method, just like a normal one, receives an automatic extra first parameter. This time, it isn't `self`, it doesn't represent the object that the method was called on (because there won't be one), but the class it is part of. If we want a method that makes use of the class attribute `counter` to tell us if any fraction objects have been created yet:

```
1 ...     @classmethod
2 ...     def have_any_been_created_yet(thisclass):
3 ...         return thisclass.counter != 0
4
5 >>> reload(fra)
```

```

6 >>> fra.fraction.have_any_been_created_yet()
7     False
8 >>> a = fra.fraction(3, 7)
9 >>> fra.fraction.have_any_been_created_yet()
10    True

```

There is something very odd about the way Python loads classes. In experimenting with this, I originally started with the exact class definition as shown. After a few runs and reloads, I added a class variable to represent  $\pi$  by adding the line

```
pi = fraction(22, 7)
```

just below the `counter = 0`. Everything worked exactly as expected, over many reloads. But then I closed the Idle session and restarted it, and couldn't import `fraction` any more. It always gave an error at the definition of `pi`, saying that `fraction` has not been defined. That isn't really the Python way, functions and methods can refer to themselves without anything special, so why can't a class make use of its own constructor? Moving the definition of `pi` to below the constructor made no difference. This still mystifies me.

But it does make a useful introduction to another use of class methods, which is in implementing the "singleton" design pattern. Sometimes we want to be sure that only one object with some special meaning should ever be created. Subsequent attempts to create an identical object should not fail, they should just return the same already-created object. `Pi` is a good example of this.

```

1 ...     pi_value = None        # a class attribute
2
3 ...     @classmethod
4 ...     def pi(thisclass):
5 ...         if thisclass.pi_value == None:
6 ...             thisclass.pi_value = fraction(22, 7)
7 ...         return thisclass.pi_value
8
9 >>> x = fra.fraction.pi()
10 >>> x
11     fraction(22, 7)
12 >>> y = fra.fraction.pi()
13 >>> id(x)
14     2737869046864
15 >>> id(y)
16     2737869046864        # the two ids are identical

```

## ii. Static methods

Creating a static method is just the same, but you say `@staticmethod` instead of `@classmethod`, and there is no special first parameter because static methods supposedly can't access anything of the class they belong to. The fact that this is untrue is illustrated in the example below, with a static method that tells us whether or not the special `pi` value has been created yet. There is also a static method that shows how they are meant to be used. It tells you whether or not a particular number would make a good denominator. Everyone knows that the

denominator of a fraction must not be zero, that knowledge requires no access to the fraction class at all.

```
1 ...     @staticmethod
2 ...     def has_pi_been_created():
3 ...         return fraction.pi_value != None
4
5 ...     @staticmethod
6 ...     def good_denominator(n):
7 ...         return return n != 0
8
9 >>> x = fra.fraction.has_pi_been_created()
10     False
11 >>> fra.fraction.pi()
12     fraction(22, 7)
13 >>> x = fra.fraction.has_pi_been_created()
14     True
15 >>> fr.fraction.good_denominator(0)
16     False
```

### iii. Methods acting as attributes

One more thing. Users who have not seen the implementation might imagine that `magnitude` should be an attribute, and be surprised that it needs a pair of parentheses. If you want to cater to that expectation, the `@property` symbol (those `@` things are supposed to be called decorators) can make a parameterless (apart from `self` of course) method seem like an attribute:

```
1 ...     @property
2 ...     def magnitude(self):
3 ...         return abs(self.top / self.bot)
4
5 >>> a = fra.fraction(22, 7)
6 >>> a.magnitude
7     3.142857142857143
```

Those users may be surprised to find that an assignment to that attribute, `a.magnitude = 66`, causes an error. If you want to allow what seems like an assignment to `magnitude`, there is a special decorator for that too:

```
1 ...     @magnitude.setter
2 ...     def magnitude(self, value):
3 ...         self.top = int(value)
4 ...         self.bot = 1
5
6 >>> a = fra.fraction(22, 7)
7 >>> a.magnitude = 66
8 >>> a
9     fraction(66, 1)
10 >>> a.magnitude
11     66
```

That isn't a very practical example, but it does show everything that is needed. The decorator is specific to the property you are dealing with, and the method that implements it must have the same name as the property too.

There is a much neater way of doing this. You just define normal methods, with whatever names you want, to define what happens in any or all of these situations: getting the attribute (`x = a.magnitude`), setting the attribute (`a.magnitude = 17`), and trying to delete the attribute (`del a.magnitude`). Then create a class variable with the name you want the property to have, and set its value to the result of the `property` function. It has four parameters, all of them optional (default is `None`), they are, in order, getter, setter, deleter, and documentation string. Any parameter of `None` means that the corresponding operation is forbidden.

```
1 ...     def get_mag(self):
2 ...         return abs(self.top / self.bot)
3 ...
4 ...     def change_mag(self, value):
5 ...         self.top = int(value)
6 ...         self.bot = 1
7 ...
8 ...     magnitude = property(get_mag, change_mag)
9
10 >>> a = fra.fraction(22, 7)
11 >>> a.magnitude
12     3.142857142857143
13 >>> a.magnitude = 17
14 >>> a
15     fraction(17, 1)
```

#### iv. Creating decorators

You can even make your own decorators for ordinary functions. The decorator is a function that takes as its parameter the function that it is to modify. It must return as its result some other function. Usually this returned function will be an inner function that as well as doing some other things, calls the original function. The name of this decorator function, coupled with the usual `@`, is the decorator symbol.

As a simple example, perhaps we have a function that is being called somewhere unexpectedly, but we can't work out where. We would want to make it traceable, so that something gets printed every time it is called or returns. Of course, we could just add some `print()`s to the function itself, but it's possible we could miss one of the returns, and this is only an example to show how it's done anyway.

```
1 >>> def traceable(f):
2 ...     def inner(x, y):
3 ...         print("The function", f.__name__, "being called")
4 ...         result = f(x, y)
5 ...         print("The function", f.__name__, "exitted")
```

```

6 ...     return result
7 ...     return inner
8
9 >>> @traceable
10 ... def ordinary(x, y):
11 ...     a = x * y
12 ...     print("I just computed", a)
13 ...     return a + 1
14
15 >>> x = ordinary(3, 6)
16     The function ordinary being called
17     I just computed 18
18     The function ordinary exited
19 >>> x
20     19

```

Because it is decorated as `@traceable`, when `ordinary` is defined, its definition is given to the function `traceable`, and replaced by whatever `traceable` returns. So when you say `ordinary(3, 6)`, it is really `inner(3, 6)` that gets executed, and as far as `inner` is concerned, the value of `f` is the original function that `traceable` had when `inner` was created. The `__name__` is nothing special. Every function knows its own name (if it has one) and that is how it is found.

Home made decorators are not particularly impressive, they can't do anything to the function while it is running, but they are moderately popular, so they are at least worth knowing about.

## 24. Inheritance

Inheritance in Python is very much as it is in other languages. If a class already exists that does a lot of what you want, but not all of it, you can build another class on top of its functionality, adding new things, and *inheriting* everything it has too. The new class is called a subclass, the one it was built from is called a superclass.

For an example, I want something that behaves exactly like an ordinary dictionary object, but has two extra methods. One is like `.keys()`, but delivers all the keys in sorted order. The other delivers the sum of all the even numeric values. To make a class inherit from another, just put the superclass name in parentheses after the subclass name, then just define the new abilities in the normal way.

```

1 >>> class specialdict(dict):
2
3 ...     def orderedkeys(self):
4 ...         k = self.keys()
5 ...         return sorted(k)
6
7 ...     def evensum(self):
8 ...         sum = 0
9 ...         for v in self.values():

```

```

10 ...         if type(v) == int and v % 2 == 0:
11 ...             sum += v
12 ...         return sum
13
14 >>> d = specialdict()
15 >>> d.update({"cat": 8, "horse": "??", "dog": 6, "xx": 17})
16 >>> d.orderedkeys()
17     ['cat', 'dog', 'horse', 'xx']
18 >>> d.evensum()
19     14

```

Notice that we could say `self.keys()` and `self.values()` even though we had not defined them, they were inherited and are part of `self`.

This is also the way to create hashable versions of non-hashable things like lists. Only hashable things can be stored in sets or used as keys in a dictionary. An object is hashable if it has a `__hash__` method that returns an int, called the object's hash value. A hash value should be quite large and seemingly random, but it is required that the hash values of two identical objects are equal. It is also important that the hash values of two unequal objects should be very unlikely to be equal.

The purpose is that a hash value (or a hash value reduced to a much smaller range through the `%` operator) can be used as a position in a list where something can be stored. Later when you need to search for an object, its hash value will tell you exactly where in the list it must be if it is there at all. It is possible for two different objects to have the same hash value, so it will really be a list of lists. `table[h]` is a list of all inserted objects whose hash value is `h`. A normal search of `table[h]` will be needed to verify presence. If an object such as a list has many parts, it is perfectly reasonable to combine the hash values of the object's parts to produce the object's own hash value. The built-in function `hash` takes any hashable object and returns its hash value.

```

1 >>> class hashable_list(list):
2 ...     def __hash__(self):
3 ...         h = 1
4 ...         for item in self:
5 ...             h = (h * hash(item)) & 0x7FFFFFFFFFFFFFFF
6 ...         return h
7
8 >>> x = hashable_list(["bat", "cat", "dog"])
9 >>> x
10    ['bat', 'cat', 'dog']
11 >>> hash(x)
12    4023728796228314549
13 >>> x.append("cat")
14    hash(x)
15 >>> 760896232467004203

```

The `& 0x7FFFFFFFFFFFFFFF` is to ensure that the numbers don't get too big, that would make the calculation very inefficient. Python uses small numbers as their own hash values, that will explain some surprise results you might find while experimenting.

It is important to remember why lists and sets are not already hashable. If you use something as a key in a dictionary and then modify it, its hash value will change, and you won't be able to find it any more.

## ii. Introspection

There are some useful things that can be done to examine an object. The `type` function works in a helpful way:

```
1 >>> type(d)
2     <class 'specialdict'>
3 >>> type(d) == specialdict
4     True
```

But always remember that you normally import your classes from a module, so you would be saying something like `type(d) == modulename.specialdict` instead.

The functions `isinstance(obj, cls)` and `issubclass(sub, sup)` do what their names suggest. `isinstance` includes superclasses in its decision making.

```
1 >>> isinstance(d, specialdict)
2     True
3 >>> isinstance(d, dict)
4     True
5 >>> isinstance(d, str)
6     False
7 >>> issubclass(specialdict, dict)
8     True
9 >>> issubclass(specialdict, object)
10    True      # because all classes are
```

## iii. Changing behaviours

Now suppose that I want to modify the behaviour of something that was inherited. If I want to completely replace the old behaviour, then I just define a new method with the same name and that's that. But if I want to make use of the old behaviour in defining the new, I can still call the old method by saying the name of the superclass instead of `self` in the call. If I do that, I also have to provide `self` explicitly as a first parameter. Let's say I want to change values so that the values are all turned into strings that are duplicated:

```
1 ...     def values(self):
2 ...         vs = []
3 ...         for v in dict.values(self):
4 ...             vs.append(str(v) * 2)
5 ...         return vs
6
7 >>> d = specialdict()
8 >>>
```

```

9 >>> d.update( {"cat": 8, "horse": "??", "dog": 6, "xx": 17 } )
10     d.values()
    ['88', '????', '66', '1717']

```

And I didn't need to be so clumsy in creating `d`. In Python, constructors are inherited along with all the other methods, so I could have made use of the `dict` constructor's ability to copy an existing `dict`:

```

1 >>> e = specialdict( {"cat": 8, "horse": "??", "dog": 6 } )
2 >>> e.values()
3     ['88', '????', '66']

```

Of course, if I want to make my own constructor but make use of the inherited one to do most of the work, that's easy.

```

1 ...     def __init__(self, d):
2 ...         print("Creating a specialdict")
3 ...         dict.__init__(self, d)

```

But if the inherited constructor has a complicated definition to allow it to take many different forms of parameters, I would have to duplicate its parameters in my own definition. Referring back to the section on functions, subsection on keyword and unknown parameters, you can make a constructor that can take any number of parameters, look at their types, and decide exactly what actions are required in each case.

```

1 >>>     def __init__(self, * a, ** k):
2 ...         ...

```

#### iv. The `super()` notation

Inside a method, the function `super()` will return a special instance of the method's class' superclass that has access to all of the current instance's data. That means that calling a method on `super()` is an alternative to explicitly saying the superclass name in the method call. It also removes the need for the `self` parameter. That means that the constructor and values methods above could have been written thus:

```

1 ...     def __init__(self, d):
2 ...         print("Creating a specialdict")
3 ...         super().__init__(d)
4
5 ...     def values(self):
6 ...         vs = []
7 ...         for v in super().values():
8 ...             vs.append(str(v) * 2)
9 ...         return vs

```

In this, the parameterless call to `super()` is shorthand for `super(specialdict, self)`. `super` takes a class, which must be a subclass of something, and an object



of that given class. The type of the object it returns, is the given class' immediate superclass. The two parameter `super` can be used outside of methods, like this:

```
1 >>> ob = specialdict({"cat": 8, "horse": "??", "dog": 6})
2 >>> sup = super(specialdict, ob)
3 >>> sup.values()
4     dict_values([8, '??', 6])
```

From the values printed, you can see that this really was a call to `dict`'s original `values()` method, they haven't been doubled. Having to explicitly state the name of the superclass that you want is missing the point, and some of the advantages, of object oriented programming.

## v. Multiple inheritance

It is possible that two different classes provide useful functionality and you want to create a class that inherits from both of them. In fact, we'll make it a little bit more complicated than that.

Class A has a method called `meth`.

Class B inherits from A and also has its own method called `meth`.

Class C has a method called `meth`.

Class D inherits from B and C and also has its own method called `meth`.

```
1 >>> class A:
2 ...     def meth(self, x):
3 ...         return 10 * x
4
5 >>> class B(A):
6 ...     def meth(self, x):
7 ...         return 100 * x
8
9 >>> class C:
10 ...     def meth(self, x):
11 ...         return 1000 * x
12
13 >>> class D(B, C):
14 ...     def meth(self, x):
15 ...         return 10000 * x
16 ...     def use(self, x):
17 ...         return [ A.meth(self, x), B.meth(self, x),
18 ...                 C.meth(self, x), D.meth(self, x) ]
19
20 >>> obj = D()
21 >>> obj.use(9)
22     [90, 900, 9000, 90000]
```

In that example we explicitly said which of our inherited class' version of `meth` was wanted. You can always do that to make things clear, but you'd be giving yourself extra work and losing flexibility. What would happen is we just said `self.meth(x)` and perhaps not all of the classes had their own `meth` to offer. Which `meth` would be chosen? The search order is very clear. When searching for a method, the current class is always checked first, then for each of the classes it

inherited, in left to right order (in our case, B before C) its entire inheritance tree is searched, lowest (B) first, highest (C) last. Last of all `object` is checked because everything inherits ultimately from `object`. To check, you can look at the class D's `__mro__` attribute. It tells us that the search order is D, B, A, C, `object`:

```
1 >>> D.__mro__
2     (<class 'multi.D'>, <class 'multi.B'>,
3      <class 'multi.A'>, <class 'multi.C'>,
4      <class 'object'>)
```

## 25. Iterables and iterators

Iterables are things like lists and tuples, as described in that section. They provide a collection of data items (objects) that can be accessed one-by-one. Sometimes the objects don't actually exist until it is their turn to be produced by the iterable. Sets dictionaries strings, bytes, bytarrays, files, and ranges are all iterables.

### i. Standard operations on iterables

The names of many iterable classes: `tuple`, `list`, `set`, `dict`, may be used to build an iterable of that type from another iterable provided as a parameter, containing the same values in the same order.

```
1 >>> tuple(range(10, 3, -1))
2     (10, 9, 8, 7, 6, 5, 4)
3 >>> list(range(10, 3, -1))
4     (10, 9, 8, 7, 6, 5, 4)
5 >>> set(range(10, 3, -1))
6     {4, 5, 6, 7, 8, 9, 10}
7 >>> tuple("elephant")
8     ('e', 'l', 'e', 'p', 'h', 'a', 'n', 't')
```

The functions `sum` (combine all the objects with the `+` operator and return the result), `max` (use the `<` or `>` operator, whichever exists, to find the biggest and return it), `min` (use the `<` or `>` operator, whichever exists, to find the smallest and return it), `any` (True if any of the items would be considered true as a condition, False if not), `all` (True if all of the items would be considered true as a condition, False if not), should work on all iterables.

`sorted` should return a list containing all the iterable's contents, in order according to the `<` or `>` operator. `sorted` may take two keyword parameters, `key =` a function that is applied to each of the items to provide the key for sorting, and `reverse = True` to reverse the order of the sorting.

```
1 >>> sorted("elephant")
2     ['a', 'e', 'e', 'h', 'l', 'n', 'p', 't']
3 >>> sorted(["Zebra", "cat", "Dog", "Yeti", "gnu"])
4     ['Dog', 'Yeti', 'Zebra', 'cat', 'gnu']
5 >>> sorted(["Zebra", "cat", "Dog", "Yeti", "gnu"],
```

```

6 ...         key = str.lower)
7     ['cat', 'Dog', 'gnu', 'Yeti', 'Zebra']
8 >>> min("elephant")
9     'a'

```

A for loop should work on any iterable, as should comprehensions.

```

1 >>> for c in "elephant":
2 ...     print(ord(c), c)
3     101 e
4     108 l
5     101 e
6     112 p
7     104 h
8     97 a
9     110 n
10    116 t
11 >>> { c * 2 for c in "elephant" if c < "m" }
12     {'ll', 'hh', 'aa', 'ee'}

```

`enumerate` creates a modified version of an iterable. It has all the same values in the same order, but they are paired with their positions in tuples. The value returned by `enumerate` is considered "opaque", you can't look at it and see the contents, but you can still produce them in all the normal ways.

```

1 >>> e = enumerate("elephant")
2 >>> e
3     <enumerate object at 0x00000210F46920C0>
4 >>> for i in e:
5 ...     print(i)
6     (0, 'e')
7     (1, 'l')
8     (2, 'e')
9     (3, 'p')
10    (4, 'h')
11    (5, 'a')
12    (6, 'n')
13    (7, 't')

```

`filter` and `map` apply a function to every element of an iterable. `filter` produces a new iterable containing all the items for which the function returns true, `map` produces a new iterable containing the results of applying the function to all of the original items. A function of three (for example) parameters may be mapped onto three separate iterators to produce an iterable of the results. If `map`'s iterables are not all of the same length, it stops as soon as any one of them runs out of items.

```

1 >>> def nice(n):
2 ...     return n % 2 == 0 and n != 6
3 >>> x = filter(nice, [ 1, 8, 3, 5, 6, 4, 2, 3, 10 ])
4 >>> x
5     <filter object at 0x00000210F46A1600>
6 >>> list(x)
7     [8, 4, 2, 10]
8 >>> def change(x):

```

```

9 ...     return x * 10 + 3
10 >>> list(map(change, [ 1, 8, 3, 5, 6, 4, 2, 3, 10 ]))
11     [13, 83, 33, 53, 63, 43, 23, 33, 103]
12 >>> a = [ 3, 7, 2, 5, 8, 4, 1 ]
13 >>> b = [ 7, 4, 3, 1, 3, 5, 6 ]
14 >>> c = "ABCDEFG"
15 >>> def combine(x, y, z):
16 ...     return z + str(x * y)
17 >>> tuple(map(combine, a, b, c))
18     ('a21', 'b28', 'c6', 'd5', 'e24', 'f20', 'g6')

```

`zip` takes any number of iterables and returns a new iterable that produces tuples. The first will contain the first items from each of the input iterables, and so on.

```

1 >>> a = [ 3, 7, 2, 5, 8, 4, 1 ]
2 >>> b = [ 7, 4, 3, 1, 3, 5, 6 ]
3 >>> c = "ABCDEFG"
4 >>> list(zip(a, b, c))
5     [(3, 7, 'A'), (7, 4, 'B'), (2, 3, 'C'), (5, 1, 'D'),
6     (8, 3, 'E'), (4, 5, 'F'), (1, 6, 'G')]

```

## ii. Iterators

An iterator is usually another kind of iterable. Like all iterables, they can be converted into other iterables and operated upon as above. But that isn't their normal use. Some are never-ending, converting them to lists or tuples would tie up your computer until you close the Idle session. The usual plan for an iterator is to just take the next item from it whenever you need it. The `next(it)` function will return the next item from the iterator, and advance it so that the next time `next` is used, you'll get the following item as expected.

The `itertools` module provides a lot of iterator-related things. The most basic is `count()`. `count()` produces an unending stream of numbers starting with 0. `count(n)` starts the sequence at `n` instead of 0. `count` normally adds one each time to get the next value, but `count(n, a)` adds `a` instead. A `count` iterator is not opaque, its `__repr__` method shows its current state.

```

1 >>> i = itertools.count(9, -2)
2 >>> next(i)
3     9
4 >>> next(i)
5     7
6 >>> next(i)
7     5
8 >>> i
9     count(5, -2)

```

`iter(iterable)` creates an iterator that produces all the elements of the iterable, in the same order.

```

1 >>> i = iter([8, "cat", (3, 0, 5), 14])

```

```

2 >>> next(i)
3     8
4 >>> next(i)
5     'cat'
6 >>> next(i)
7     (3, 0, 5)
8 >>> next(i)
9     14
10 >>> next(i)    # causes an error

```

`repeat(x, n)` is an iterator that produces the same value `n` times, and then is exhausted. If `n` is not provided, it just goes on for ever.

```

1 >>> i = itertools.repeat("cat", 3)
2 >>> next(i)
3     'cat'
4 >>> next(i)
5     'cat'
6 >>> next(i)
7     'cat'
8 >>> next(i)
9     Traceback ... ..
10    StopIteration

```

This gives us an opportunity to see how to detect the end of an iterator without having to use a `try` statement. `next` can take a second parameter, which provides the value that will be returned instead of raising an exception if the iterator is exhausted. To be completely safe from false positives, you can create a unique object for this value:

```

1 >>> done = object()
2 >>> i = itertools.repeat("cat", 3)
3 >>> while True:
4 ...     x = next(i, done)
5 ...     if x is done:
6 ...         break
7 ...     print(x)
8     cat
9     cat
10    cat

```

`combinations(iterable, n)` produces all possible `n`-tuples from the iterable maintaining the order of the elements. Note that in this example below, the order of the input is 4, 2, 3, 1; every tuple that begins with 4 is produced withs, then all the 2s, then all the threes, and within the 4s and 2s the item in the second position follows the same order.

`permutations` produces *all* possible combinations. They are also produced in order, but `permutations` does not consider tuples to be duplicates just because they have the same content but in a different order. `permutations` will produce both (1, 4) and (4, 1), `combinations` would only produce (1, 4).

`combinations` never pairs a value with itself (unless that value appears in the iterable twice) but `combinations_with_replacement` does.

`product` is a bit like combinations but it takes any number of iterables and groups their contents in all combinations into tuples. It is really making a Cartesian product. If you want to combine an iterable with itself a number of times, just provide the iterable once, and use the keyword parameter `repeat`.

```

1 >>> list(itertools.combinations((4, 2, 3, 1), 2))
2     [(4, 2), (4, 3), (4, 1), (2, 3), (2, 1), (3, 1)]
3
4 >>> list(itertools.permutations((4, 2, 3, 1), 2))
5     [(4, 2), (4, 3), (4, 1), (2, 4), (2, 3), (2, 1),
6      (3, 4), (3, 2), (3, 1), (1, 4), (1, 2), (1, 3)]
7
8 >>> list(itertools.combinations_with_replacement((4, 2, 3, 1), 2))
9     [(4, 4), (4, 2), (4, 3), (4, 1), (2, 2), (2, 3),
10      (2, 1), (3, 3), (3, 1), (1, 1)]
11
12 >>> list(itertools.product([1, 2, 3], ("a", "b"), [9, 8]))
13     [(1, 'a', 9), (1, 'a', 8), (1, 'b', 9), (1, 'b', 8),
14      (2, 'a', 9), (2, 'a', 8), (2, 'b', 9), (2, 'b', 8),
15      (3, 'a', 9), (3, 'a', 8), (3, 'b', 9), (3, 'b', 8)]
16
17 >>> list(itertools.product(["a", "b"], repeat = 3))
18     [('a', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a'),
19      ('a', 'b', 'b'), ('b', 'a', 'a'), ('b', 'a', 'b'),
20      ('b', 'b', 'a'), ('b', 'b', 'b')]

```

`tee(iterable, n)` returns an `n`-tuple of iterators based on the iterable. All are initially identical but independent. If `n` is not provided, it defaults to 2. `(x, y) = tee([4, 2, 7, 3])` could give us this sequence: `next(x) ⇒ 4, next(x) ⇒ 2, next(x) ⇒ 7, next(y) ⇒ 4, next(y) ⇒ 2, next(x) ⇒ 3, next(y) ⇒ 7, next(y) ⇒ 3`.

`cycle(iterable)` just produces the values from the given iterable over and over again. `cycle([2, 6, 3])` would produce 2, 6, 3, 2, 6, 3, 2, 6, 3, 2, ... unendingly.

`chain` appends a bunch of iterables, and produces all of their values just once before becoming exhausted. `chain("cat", "dog", [7, 5, 9], "bat")` produces 'cat', 'dog', 7, 5, 9, 'b', 'a', 't'.

`islice(it, na, nb, nc)` does the work of a slice, but on an iterator. If only `na` is provided, it produces the first `na` values from `it`, then stops. If only `na` and `nb` are provided, it produces the items between positions `na` and `nb - 1` inclusive, just like the slice `s[na:nb]`. If all three are provided then after producing each item, it will skip over then next `nc - 1` to find the next.

`islice(count(10), 5)` produces 10, 11, 12, 13, 14.

`islice(count(10), 5, 15)` produces 15, 16, 17, 18, 19, 20, 21, 22, 23, 24

`islice(count(10), 5, 15, 3)` produces 15, 18, 21, 24

`accumulate(iterable, f)`, `f` must be a two parameter function. The first value produced is `iterable[0]`. After that `f` is used on the previous value produced and

the next value in the iterable to produce the next result. The keyword parameter `initial` lets you prime the results with a value that is not in the iterable.

```
1 >>> list(itertools.accumulate([1, 2, 3, 4, 5], operator.add))
2     [1, 3, 6, 10, 15]
3 >>> list(itertools.accumulate([1, 2, 3, 4, 5], operator.mul))
4     [1, 2, 6, 24, 120]
5 >>> list(itertools.accumulate(["ab", "cde", "f", "gh"],
6 ...                             operator.add))
7     ['ab', 'abcde', 'abcdef', 'abcdefg', 'abcdefgh']
8 >>> tuple(itertools.accumulate([1, 2, 3, 4, 5],
9 ...                             operator.add, initial = 100))
10    (100, 101, 103, 106, 110, 115)
```

`pairwise(iterable)` produces the list of all possible neighbouring pairs from the iterable. `pairwise("horse")` produces `('h', 'o')`, `('o', 'r')`, `('r', 's')`, `('s', 'e')`.

`compress` takes an iterable of anythings and a same-length iterable of bools. The values produced are those from the first iterable that correspond with Trues in the second.

```
1 >>> list(itertools.compress(
2 ...     ("Sun", "Mon", "Tue", "Wed", "Thur", "Fri", "Sat"),
3 ...     (False, True, True, True, False, True, False)))
4     ['Mon', 'Tue', 'Wed', 'Fri']
```

`filterfalse`, `takewhile`, and `dropwhile` take a function and an iterable. The values produced are those from the iterable that are in some way selected by the function.

`filterfalse` produces only the items for which the function returns `False`. Why would they choose to filter on `False`? You return `False` to indicate yes? Filtering is always done on `True`.

`takewhile` produces items for as long as the function always returns `True` for them. As soon as an item gets a `False`, the iterator is finished.

`dropwhile` ignores items for as long as the function always returns `True` for them. As soon as an item gets a `False`, it and all subsequent items will be produced by the `dropwhile` iterator, the function is ignored from then on.

```
1 >>> list(itertools.filterfalse(
2 ...     lambda x: x > 6 and x < 15,
3 ...     itertools.islice(itertools.count(1), 21)))
4     [1, 2, 3, 4, 5, 6, 15, 16, 17, 18, 19, 20, 21]
5 >>> list(itertools.takewhile(
6 ...     lambda x: x % 2 == 0,
7 ...     [4, 22, 8, 94, 6, 9, 2, 4, 6, 8, 11]))
8     [4, 22, 8, 94, 6]
9 >>> list(itertools.dropwhile(
10 ...     lambda x: x % 2 == 0,
11 ...     [4, 22, 8, 94, 6, 9, 2, 4, 6, 8, 11]))
12    [9, 2, 4, 6, 8, 11]
```

`starmap` takes a function and an iterable of tuples. Its values are produced by giving the function each of the tuples as its parameter list.

```
1 >>> list(itertools.starmap(
2 ...     operator.add,
3 ...     [(2, 5), (10, 3), ("a", "nt"), (math.pi, 1)]))
4 [7, 13, 'ant', 4.141592653589793]
5 >>> list(itertools.starmap(
6 ...     lambda x, y, z: z + y + x,
7 ...     [(1, 2, 3), (8, 2, 5), ("a", "b", "c"), (0, 0, 7)]))
8 [6, 15, 'cba', 7]
```

`zip_longest` is like the ordinary `zip` function for iterables, except that whereas that function stops as soon as any of the iterables is exhausted, `zip_longest` continues until all of them are exhausted. `None` is used to take the place of the value from an exhausted iterable. The keyword argument `fillvalue` allows you to specify a value other than `None`.

```
1 >>> list(itertools.zip_longest(
2 ...     (9, 8, 7),
3 ...     itertools.islice(itertools.count(10), 6),
4 ...     "abcd"))
5 [(9, 10, 'a'), (8, 11, 'b'), (7, 12, 'c'),
6  (None, 13, 'd'), (None, 14, None), (None, 15, None)]
7 >>> list(itertools.zip_longest(
8 ...     (9, 8, 7),
9 ...     itertools.islice(itertools.count(10), 6),
10 ...     "abcd",
11 ...     fillvalue = "o"))
12 [(9, 10, 'a'), (8, 11, 'b'), (7, 12, 'c'),
13  ('o', 13, 'd'), ('o', 14, 'o'), ('o', 15, 'o')]
```

### iii. Making an iterator

What is the real difference between an iterable and an iterator? Most of what appears on-line from the Python community is very confused on this issue, but there is a very simple answer: Any object that has an `__iter__` method is, by definition, an iterable. Any object that has a `__next__` method is by definition an iterator. The `__iter__` method creates an iterator from an iterable. The `__next__` method just returns the next result from an iterator. It is possible, and quite common, for an object to be both iterable and iterator, but that sometimes leads to trouble.

Once you make an iterator out of an object, everything is easy. Every time `next(obj)` is used, `obj`'s `__next__` method is called to produce the result. If you decide that the iterator is exhausted, raise the `StopIteration` exception.

The `__iter__` method is like a constructor for an iterator. `iter(obj)` calls `obj`'s `__iter__` method, which does whatever setup is required for an iterator and we are usually advised that it should return `self`. But doing that causes trouble. If you create an object and then use `iter` twice to get two iterators of that type, the two iterators and the original object will all be the same thing. Not just identical, but the same, there is only one object there. Saying `next()` to the first



iterator will advance the second one too, and that is almost certainly not the behaviour that would be expected or wanted. Even if you only create one iterator from the original object, you will still find that everything you do to the iterator is also done to the original object.

This example shows a very inefficient iterator that produces prime numbers. It behaves like an iterator, but is not one, due to the definition above. It has a `get_next_prime` instead of a proper iterator's `__next__`.

```
1 >>> class primemaker:
2 ...
3 ...     def __init__(self):
4 ...         self.value = 2
5 ...
6 ...     def am_i_prime_now(self):
7 ...         for i in range(2, self.value):
8 ...             if self.value % i == 0:
9 ...                 return False
10 ...         return True
11 ...
12 ...     def get_next_prime(self):
13 ...         while True:
14 ...             if self.am_i_prime_now():
15 ...                 result = self.value
16 ...                 self.value += 1
17 ...                 return result
18 ...                 self.value += 1
19
20 >>> pm = primemaker()
21 >>> for i in range(20):
22 ...     print(pm.get_next_prime(), end = ", ")
23     2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
24     47, 53, 59, 61, 67, 71,
```

It is easy to make it into both an iterable and its own iterator. Just add the two required methods. This version follows the advice generally given. The existing parts of the class definition will not be repeated.

```
1 ...     def __iter__(self):
2 ...         return self
3 ...
4 ...     def __next__(self):
5 ...         return self.get_next_prime()
```

But we can easily see the problem. In the following, two iterators seem to be created, but it is clear they are both the same thing. Calling on each of the iterators at random, we don't see the repeats that would be expected. Advancing `it_one` also advances `it_two` because they are the same object.

```
1 >>> pm = primemaker()
2 >>> it_one = iter(pm)
3 >>> it_two = iter(pm)
4 >>> for i in range(20):
5 ...     if random.randint(1, 2) == 1:
6 ...         print("one:", next(it_one), sep = "", end = " ")
```

```

7 ...     else:
8 ...         print("two:", next(it_two), sep = "", end = " ")
9     one:2 one:3 one:5 one:7 two:11 two:13 one:17 one:19 one:23 two:29
10    one:31 two:37 two:41 one:43 two:47 two:53 two:59 one:61 two:67
11    two:71

```

One simple but not totally successful way to handle this would be to make `__iter__` return a totally new object each time it is called.

```

1 ...     def __iter__(self):
2 ...         return primemaker()

```

The same code now produces two clearly independent iterators:

```

one:2 one:3 two:2 two:3 one:5 one:7 one:11 one:13 two:5
one:17 one:19 one:23 one:29 one:31 one:37 one:41 two:7
one:43 one:47 two:11

```

But that victory is very short lived. Think back to `itertools.islice(it, n)`, which gives you `n` values from the iterator `it`. I'll create an iterator, take a few values from it, then use `islice` to get ten more:

```

1 >>> pm = primemaker()
2 >>> it = iter(pm)
3 >>> (next(it), next(it), next(it), next(it), next(it))
4     (2, 3, 5, 7, 11)
5 >>> tuple(islice(it, 10))
6     (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)

```

It takes a bit of thinking to see how that happened. How could the iterator go right back to the beginning? Do iterators keep all the values they've produced so you can repeat them? Do iterators store their initial state so they can be reset? No. Both of those ideas would be quite bad. And why would anyone want to restart an iterator anyway? The reality is that `islice` needs an iterator, so whenever it is given an iterable, it calls its `__iter__` to make an iterator without even checking to see if it already is one. I don't call that sensible. If you want an iterator and you're given an iterator, just use it. But there are no hard and fast rules about this sort of thing.

We could create our own versions of all the `itertools` functions and make them treat iterators with a bit more respect. Or perhaps we can find a less wasteful approach. Maybe `__iter__` can still create a new object, but also change that new object's state to be identical to the current object. That would be messy for a more complex object, but we'll try it out. `primemaker`'s entire state is in the single variable `value`.

```

1 ...     def __iter__(self):
2 ...         result = primemaker()
3 ...         result.value = self.value
4 ...         return result
5
6 >>> pm = primemaker()

```

```

7 >>> it = iter(pm)
8 >>> (next(it), next(it), next(it), next(it), next(it))
9     (2, 3, 5, 7, 11)
10 >>> tuple(islice(it, 10))
11     (13, 17, 19, 23, 29, 31, 37, 41, 43, 47)
12 >>> (next(it), next(it), next(it), next(it), next(it))
13     (13, 17, 19, 23, 29)

```

A partial success, but of course when `islice` finishes, we're back to the original object, with no way to recover `islice`'s private iterator's final state.

Just to get one detail out of the way: having `__iter__` make a new object and try to manipulate its state is not very satisfactory. An iterable's state could be very complex. It would be much more straight-forward and generalisable just to make a deep copy instead:

```

1 ...     def __iter__(self):
2 ...         return copy.deepcopy(self)

```

The next step towards a solution could be to have two different kinds of object. Our `primemaker` would be an iterable, but not an iterator (i.e. it has an `__iter__` but no `__next__`). `primemaker`'s `__iter__` would create a totally new but trivial object to be the iterator. The new object would take a deep copy of the original `primemaker` (that's much more general-purpose than creating a new `primemaker` and trying to make its state match the original's exactly). That deep copy would do all the real work.

The real trick to this is that the new object's `__iter__` would go back to the old design and just return `self`. When we deliberately create an iterator from a `primemaker` we do create a new object, so all of its iterators are independent. But when `islice` tries to make a new iterator out of it, it doesn't. `islice`'s iterator is the same object as our intentional iterator.

In `primemaker`:

```

1 ...     def __iter__(self):
2 ...         return primeiterator(self)

```

And the new `primeiterator`:

```

1 >>> class primeiterator:
2 ...
3 ...     def __init__(self, maker):
4 ...         self.worker = copy.deepcopy(maker)
5 ...
6 ...     def __iter__(self):
7 ...         return self
8 ...
9 ...     def __next__(self):
10 ...         return self.worker.get_next_prime()
11
12 >>> pm = primemaker()

```

```

13 >>> it = iter(pm)
14 >>> (next(it), next(it), next(it), next(it), next(it))
15     (2, 3, 5, 7, 11)
16 >>> tuple(islice(it, 10))
17     (13, 17, 19, 23, 29, 31, 37, 41, 43, 47)
18 >>> (next(it), next(it), next(it), next(it), next(it))
19     (53, 59, 61, 67, 71)

```

If making a deep copy is satisfactory, then the new class to represent an iterator is unnecessary. So long as it remembers whether it is the result of `__iter__` or not, `primemaker` could do the entire job itself. It would of course have to have its `__next__` restored.

In `primemaker`:

```

1 ...     def __init__(self):
2 ...         self.value = 2
3 ...         self.i_came_from_an_iter = False
4 ...
5 ...     def __next__(self):
6 ...         return self.get_next_prime()
7 ...
8 ...     def __iter__(self):
9 ...         if self.i_came_from_an_iter:
10 ...             return self
11 ...         else:
12 ...             result = copy.deepcopy(self)
13 ...             result.i_came_from_an_iter = True
14 ...             return result

```

## 26. Generators

A generator is a lot like an iterator, but it is implemented as a function, not a class. Using the `yield` statement, a function is able to pause in its execution, returning control back to its caller, but then be woken up again later, and continue undisturbed from exactly where it `yielded`. This is a use of a closure. A closure is simply an object that stores all the information necessary for that to work. It includes of course which function was running, exactly where in that function it had reached, the values of all local variables and parameters, and a few more things too. Here is a very simple example, inefficiently generating prime numbers again, but this time limited to those less than 100.

```

1 >>> def primey():
2 ...     for n in range(1, 100):
3 ...         good = True
4 ...         for i in range(2, n):
5 ...             if n % i == 0:
6 ...                 good = False
7 ...                 break
8 ...         if good:
9 ...             yield n

```

Calling `primey` will not return a prime number. It will return one of those closure objects that Python calls a generator. On that first call, the function is sort-of started, just so there is something to make a closure of, but it is stopped before it can do anything at all. You use `next()` to get successive prime numbers out of it. Each call to `next()` lets the closure continue until it meets a `yield`. After the function exits, `next()` causes a `StopIteration` exception.

```
1 >>> it = primey()
2 >>> while True:
3 ...     print(next(it), end = " ")
4       2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
5       79 83 89 97 Traceback ...
6       ... StopIteration
```

Naturally, a `for` loop and all the other tricks work too.

```
1 >>> for n in primey():
2 ...     print(n)
```

Communication with a generator doesn't have to be one way. You can also send objects to them. You can only send something to a generator while it is at a `yield`, and you can only send one thing at a time. Each `send` will trigger it to run again until it reaches another `yield`.

To send an object to a generator, just use its `send` method. It works just like `next` in that it returns the value that was `yielded`, but it also sends its parameter. Inside the generator, just capture the value returned by `yield`. If nothing was sent, the value will be `None`.

This demonstration is a modification of the previous one. `primey` always captures and prints the value `yield` returns. `user` begins with a `next`, to ensure that it won't do an illegal `send` before the first `yield`. Then in its loop, it randomly chooses whether to send something or just use `next`. Either way, it prints the value that was `yielded`. `random.random()` produces a number between 0 and 1. `random.randint(min, max)` produces an `int` within the range given.

```
1 >>> import random
2
3 >>> def primey():
4 ...     for n in range(2, 100):
5 ...         good = True
6 ...         for i in range(2, n):
7 ...             if n % i == 0:
8 ...                 good = False
9 ...                 break
10 ...         if good:
11 ...             v = yield n
12 ...             print("primey received", v)
13
14 >>> def user():
15 ...     p = primey()
16 ...     x = next(p)
```

```

17 ...     print("user started with", x)
18 ...     for i in range(1, 8):
19 ...         if random.random() < 0.33:
20 ...             r = random.randint(9000, 9999)
21 ...             x = p.send(r)
22 ...             print("user sent", r, "and got", x)
23 ...         else:
24 ...             x = next(p)
25 ...             print("user got", x)
26
27 >>> user()
28     user started with 2
29     primey received None
30     user got 3
31     primey received 9169
32     user sent 9169 and got 5
33     primey received None
34     user got 7
35     primey received None
36     user got 11
37     primey received 9551
38     user sent 9551 and got 13
39     primey received None
40     user got 17
41     primey received None
42     user got 19

```

Notice that the sends and receives seem to be out of order. `primey` reports receiving 9169 before `user` reports sending it. That is simply because `user` prints its “sent” message after calling `send`, and control isn't returned from `send` until `primey` reaches the next `yield`.

If you need to make a generator do something unusual, and it is too much trouble to implement a proper communications protocol, you can instead trigger an exception inside the generator function. That is done by calling the generator's `throw` method, with an exception class as its parameter. So if `user` wanted to get some special information out of `primey`, maybe the value of its loop counter `i`, `user` would simply say `result = p.throw(Exception)`. That would of course only happen at the next `yield`. `primey` would have to surround every `yield` with a `try` statement like this:

```

1 ...     try:
2 ...         v = yield n
3 ...         print("primey received", v)
4 ...     except Exception:
5 ...         yield ("loop count", i)

```

With that, at the next `yield` after `user` caused the exception, `user` would receive back from the call to `throw` the tuple `("loop count", n)` as the value of the variable `result`.

If ever you know a number of next values that need to be yielded in advance, you can effectively yield them all at once by saying `yield from it`, where `it` is an

iterable. It is equivalent to saying `for i in it: yield i`, so it is only a very small convenience. But that isn't its whole point. It also lets you take the divide and conquer approach to achieving the effect of a big complicated generator from a few smaller simpler ones. The *it* that is yielded from could be a call to another generator function.

If a generator needs to be recursive, a little extra thought is required. Suppose we have built an Ordered Binary Tree whose nodes are objects from this class

```
1 >>> class node:
2 ...     def __init__(self, d):
3 ...         self.data = d
4 ...         self.left = None
5 ...         self.right = None
```

but rather than doing everything with `node` methods, we wrote separate functions for all the operations. This would be the function for printing everything in a tree in order.

```
1 >>> def print_all(t):
2 ...     if t == None:
3 ...         return
4 ...     print_all(t.left)
5 ...     print(t.data)
6 ...     print_all(t.right)
```

That example might lead us to write a generator that produces all of a tree's contents in order in the same way:

```
1 >>> def enumerate(t):
2 ...     if t == None:
3 ...         return
4 ...     enumerate(t.left)
5 ...     yield t.data
6 ...     enumerate(t.right)
```

Unfortunately, that hasn't got a chance. Every recursive call to generate just creates another generator, and those generators would only be able to `yield` their results if their caller (the original instance of `enumerate`) executed a `next` to wait for them. `yield from` provides a very tidy solution:

```
1 >>> def enumerate(t):
2 ...     if t == None:
3 ...         return
4 ...     yield from enumerate(t.left)
5 ...     yield t.data
6 ...     yield from enumerate(t.right)
```

Second order generators, that is generators that manipulate other generators, can also be created. Think back to `itertools.count`, we could easily create a generator that approximates its simplest form:

```

1 >>> def intsfrom(n):
2 ...     while True:
3 ...         yield n
4 ...         n += 1

```

We are unlikely to want to make a list of all the ints, so perhaps we could produce a generator that just gives the first few items yielded by some other generator:

```

1 >>> def first(n, other):
2 ...     for i in range(n):
3 ...         yield next(other)
4
5 >>> list(first(12, intsfrom(7)))
6     [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]

```

## 27. Saving and restoring live data

What that title means is that our program has built up some data structures that are not simple enough to be adequately stored in a text, csv, or binary file, but we need to be able to save them anyway. There is one aspect of data structures that must be understood before the problem can be addressed: are they cyclic. Any data structure that goes beyond triviality will involve objects that contain (pointers or references to) other objects. So long as those pointers all go the same way, every pointer followed gets us further from our starting point, saving and restoring is not very difficult. But if there is some path from one object back to itself, things are different. If we have 13 objects named A to M, A refers to B and C and D, B refers to E and F, C refers to G and H, D refers to I and J and K, H refers to L and M, and L refers to F and C. That is cyclic because it is possible to get from H back to H again by following the path  $H \rightarrow L \rightarrow C \rightarrow H$ .

### i. JSON - not for cyclic structures

The Javascript Object Notation is so called because the way it describes data structures looks like the way Javascript does. They are not really related. JSON provides a notation for objects (which look just like Python dictionaries), Arrays (which look just like Python lists), and basic data (string, number, true, false, null). It is usually written properly indented and aligned, with everything on a line of its own. That makes it quite good for debugging, you can see exactly what you've got at a glance.

The `json` module provides four essential class methods and a few other things to go with them. The four essential methods can take many parameters, only the important ones are given here, we'll get to the others soon.

`json.dumps(obj, indent = None)`: returns a string containing the JSON representation of the given object. `indent` really should be a moderately small int. The default value of `None` squeezes everything together in an unreadable way. A value of `n` means that every time we get one step further into a data structure, `n` extra spaces of indentation will be inserted.



`json.dump(obj, file, indent = None)`: (no 's' in the name) does the same thing, but instead of returning a string it writes the whole things to the given file. This must be an object similar to that returned by `open`, and of course must not be read only. You are responsible for opening the file first and closing it afterwards.

```
1 >>> a = { "bat": 23, "cat": 48, "dog": 6 }
2 >>> b = [ True, 3.14159, "horse" ]
3 >>> c = { "first": a, "second": b, "third": 3 }
4 >>> json.dumps(c, indent = 3)
5     '{\n  "first": {\n      "bat": 23, ... "third": 3\n}'
6 >>> # not good, we don't want the newlines to appear as \n.
7 >>> print(json.dumps(c, indent = 3))
8     {
9         "first": {
10            "bat": 23,
11            "cat": 48,
12            "dog": 6
13        },
14        "second": [
15            true,
16            3.14159,
17            "horse"
18        ],
19        "third": 3
20    }
```

`json.loads(s)`: `s` must be a string following the format of the strings produced by `json.dumps`. It returns the collections of nested objects that the string represents.

`json.load(file)`: the same, but reads a JSON string from the file instead.

```
1 >>> J = json.dumps(c, indent = 3)
2 >>> o = json.loads(J)
3 >>> type(o)
4     <class 'dict'>
5 >>> o["first"]
6     {'bat': 23, 'cat': 48, 'dog': 6}
7 >>> o["second"]
8     [True, 3.14159, 'horse']
9 >>> o["third"]
10    3
```

Now for the bad news. JSON only allows dictionary keys to be strings, any other type gets converted to a string. Even worse, “any other type” can only be `int`, `float`, `bool`, or `None`, others produce an error. Also, JSON does not understand tuples, they get converted to lists. And JSON does not understand functions or objects produced from class definitions at all.

```
1 >>> d = { 12: [3, 4, 5], "cat": 8, "dog": 6 }
2 >>> s = json.dumps(d)
3 >>> s
4
```

```

5 >>> '{"12": [3, 4, 5], "cat": 8, "dog": 6}'
6     json.loads(s)
7 >>> {'12': [3, 4, 5], 'cat': 8, 'dog': 6}
8     json.loads(json.dumps((1, 2, 3)))
9     [1, 2, 3]

```

To make your own class JSON-able, create a special class that inherits from `json.JSONEncoder` and replace its default method. When you use `dump` or `dumps`, provide your class as the `cls` keyword parameter. `JSONEncoders` do the hard work for `dump` and `dumps`, they help with converting different types to JSON strings. Their default method is called when they encounter a type that they don't know about. All it has to do is return an object that JSON does know about. The simplest solution is just to encode your object as though it were a dictionary. Every object has a `__dict__` attribute which is a dictionary that contains all the object's attribute names and values, so back to the fraction class example:

```

1 >>> half = fraction(1, 2)
2 >>> half.__dict__
3     {'top': 1, 'bot': 2}
4 >>> import json
5 >>> from json import JSONEncoder
6 >>> class encode_as_dict(JSONEncoder):
7 ...     def default(self, obj):
8 ...         return obj.__dict__
9
10 >>> print(json.dumps(half, indent = 3, cls = encode_as_dict))
11     {
12         "top": 5,
13         "bot": 2
14     }

```

That of course will work with any kind of object that was defined by a class definition. Built-in objects like tuples don't have a `__dict__` attribute. But there is a problem. Every class-defined object will be encoded in the same way. When the decoder sees a dictionary, how is it going to know what kind of object to turn it back into?

Part of the solution is to add an entry to the dictionary that you return, it should have a name that is very unlikely to appear naturally, and its value should be something, probably a string, that will tell you what it should be rebuilt as. Every object has a `__class__` attribute that tells you which class it came from, and a class object has a `__name__` attribute which the `dir` function mysteriously doesn't show.

```

1 >>> class encode_as_dict(JSONEncoder):
2 ...     def default(self, obj):
3 ...         name = obj.__class__.__name__
4 ...         extra = { "__type__": name }
5 ...         return obj.__dict__ | extra
6
7 >>> print(json.dumps(half, indent = 3, cls = encode_as_dict))
8     {
9

```

```

10     "top": 5,
11     "bot": 2,
12     "__type__": "fraction"
    }

```

Remember that what we're looking at there really is a string. It looks like a dictionary because that is what JSON does, and `print` prints things nicely. It is really just `{\n \"top\": 5,\n \"bot\": 2, \n... tion\"\n}`.

We need to use the dictionary union operator `|` rather than its `update` method because we don't want to change the object's `__dict__` permanently. If you have classes defined inside other classes, use the `__qualname__` attribute instead of `__name__`, it will give you a unique name something like `\"bigclass.littleclass\"`

Creating a `JSONDecoder` is a little more complex, and most programmers don't go that way. Our class that inherits from `JSONDecoder` must have a constructor, we need to intercept the constructor's parameters, change one of them, and then let them continue to `JSONDecoder`'s own constructor. The parameter `object_hook` needs to be a method that can take the dictionary that dumps would normally produce and return the correct object instead, that is our decoder method. The decoder method needs to make sure that it leaves the dictionary untouched if it doesn't represent an object of a class that it knows how to handle. We'll use it on the JSON string created in the previous example

```

1 >>> class decode_from_dict(JSONDecoder):
2 ...
3 ...     def __init__(self, ** parameters):
4 ...         parameters["object_hook"] = self.decoder
5 ...         super().__init__(** parameters)
6 ...
7 ...     def decoder(self, dic):
8 ...         if "__type__" in dic:
9 ...             if dic["__type__"] == "fraction":
10 ...                 return fraction(dic["top"], dic["bot"])
11 ...         return dic
12
13 >>> s = json.dumps(half, indent = 3, cls = encode_as_dict)
14 >>> o = json.loads(s, cls = decode_from_dict)
15 >>> o
16     fraction(1, 2)

```

Those who don't want to define a whole class can define an ordinary function instead. Nothing is lost in doing this. The function does exactly what the decoder method did, and is provided to `loads` or `load` as the `object_hook` keyword parameter.

```

1 >>> def decode_from_dict(dic):
2 ...     if "__type__" in dic:
3 ...         if dic["__type__"] == "fraction":
4 ...             return fraction(dic["top"], dic["bot"])
5 ...     return dic

```

```

6
7 >>> s = json.dumps(half, indent = 3, cls = encode_as_dict)
8 >>> o = json.loads(s, object_hook = decode_from_dict)
9 >>> o
10     fraction(1, 2)

```

All of this relies on the objects you are trying to reconstruct having a constructor that just takes the values for all of the attributes. This is not usually the case. And it isn't difficult to overcome. Just remember that nothing is really protected in Python. You can use the existing constructor with the parameters it does take, then just assign to the other attributes. You can add any attributes you want.

```

1 >>> seven_ninths = fra.fraction(7, 9)
2 >>> seven_ninths.colour = "blue"

```

You don't need to worry about your decoder recursively checking for dictionaries within dictionaries, `load` and `loads` take care of that for you. But you do have to worry about incomplete information. Very often an object will not save everything that was used to construct it in attributes, so you will not have the necessary information to call the constructor. Suppose that the constructor for a `fraction` took just a single string parameter, like "22/7", and extracted the initial values for `top` and `bot` from it. We would no longer be able to use the constructor to recreate the `fraction` from the `top` and `bot` values in its JSON dictionary. Well of course we could, quite easily, but this is just an example. We can get over this problem by using the `__new__` class method to bypass the constructor, and a loop to turn everything from the dictionary except for `__type__` into an attribute.

```

1 >>> def decode_from_dict(dic):
2 ...     if "__type__" in dic:
3 ...         if dic["__type__"] == "fraction":
4 ...             f = fra.fraction.__new__(fra.fraction)
5 ...             for key in dic:
6 ...                 if key != "__type__":
7 ...                     setattr(f, key, dic[key])
8 ...             return f
9 ...     return dic

```

If the circumstances are right, we can generalise this even further, and won't even need to know which class' objects might have been JSONised. If you know where a class has been imported from, and use that information when setting the `__type__` value in the `JSONEncoder`, the function `locate` can find a class object just from the class name:

```

1 >>> cla = locate("fra.fraction")
2 >>> threeq = cla.__new__(cla)
3 >>> threeq.top = 3
4 >>> threeq.bot = 7
5 >>> threeq
6     fraction(3, 7)
7 >>> locate("complex")(4, 5)
8     (4+5j)

```

Note that `locate("fraction")` wouldn't work. It isn't just going to search through every `.py` file in the world looking for that class definition. If the class was defined in the current module or is a built-in type, its name alone is enough, as you see for `complex` above. Otherwise you need to know which import the class came from. And you certainly can't locate a class from a module that you haven't imported yet.

## ii. Pickle - OK for cyclic structures

Pickle is a serialiser, it can convert any interconnected collection of objects into a stream of bytes, and then convert back again later. It keeps a record of every object it serialises in a single use, so that if the same object is encountered again because we are serialising a cyclic structure it will just be able to insert a reference to the part of the byte stream that already represents that object, and avoid the infinite recursion. Pickle can not handle lambdas, or anything that in some way represents a live connection to something, such as an open file or socket object, but I haven't found anything else to be missing.

Pickle can be dangerous. It is not difficult to create a stream of bytes that would result in any code you want being executed automatically in the unpickling process. Don't unpickle anything you find on the web.

Just like JSON, pickle is driven by four class methods: `dumps(obj)`, `dump(obj, file)`, `loads(bytes)`, and `load(file)`, and they work in the same way, just small differences in special parameters that are not normally needed, and the fact that they use `bytes` objects instead of strings.

```
1 >>> import pickle
2 >>> o = {"a": 345, "b": [4, 8, 2]}
3 >>> s = pickle.dumps(o)
4 >>> s
5     b'\x80\x04\x95\x1a\x00\x00\x00\x00 ... \x08K\x02eu.'
6 >>> p = pickle.loads(s)
7 >>> p
8     {'a': 345, 'b': [4, 8, 2]}
9 >>> o == p
10    True
11 >>> o is p
12    False
```

With `dump` and `load`, the file must have already been opened with `"wb"` or `"rb"` respectively, or an equivalent object, and you must close it afterwards. Pickle also provides two error classes so you can catch problems with a `try` statement. They are `pickle.PicklingError` for `dumps` and `dump`, and `pickle.UnpicklingError` for `loads` and `load`. They are both subclasses of `pickle.PickleError`.

## iii. Shelve - a dictionary on disc

A `shelve` object behaves just like a dictionary, except that it lives on disc, not in memory, so its contents will survive from one run of a program to the next. There are two restrictions: the keys to this dictionary may only be strings, and the values have to be things that `pickle` can handle.

A `shelve` object is created or re-accessed through the class method `open`. It takes two parameters: a string for the name, and another string (of length one) for the mode. Don't provide an extension with the name. When a new `shelve` database is created, it will usually consist of more than one file, they will all have the name you provided, but different extensions. On this computer now, they are `.bak`, `.dir`, and `.dat`. When you open an existing database, again just provide the same base name, and the correct files will be found.

The second parameter to `open` can have one of four values:

"r": the file must already exist, and it will be read-only.

"w": the file must already exist, and it will be open for both reading and writing.

"c": the file will be created if it doesn't already exist but reopened unchanged if it does. It will be open for both reading and writing.

"n": the file will be created, if it already exists its contents will be lost. It will be open for both reading and writing.

If you don't provide a second parameter, the default is "c".

You must remember to use the `close` method on a `shelve` object when you have finished with it. Apart from that, just use it as a dictionary.

```
1 >>> import shelve
2 >>> sh = shelve.open("shdb", "n")
3 >>> sh["ant"] = "hello"
4 >>> sh["bat"] = [1, 2, 3]
5 >>> sh["cat"] = { "z": 88, "y": "yak", "x": 6.41 }
6 >>> sh["bat"]
7     [1, 2, 3]
8 >>> del sh["bat"]
9 >>> "bat" in sh
10    False
11 >>> "cat" in sh
12    True
13 >>> for key in sh:
14 ...     print(key, sh[key])
15     ant hello
16     cat {'z': 88, 'y': 'yak', 'x': 6.41}
17 >>> list(sh.keys())
18     ['ant', 'cat']
19 >>> list(sh.values())
20     ['hello', {'z': 88, 'y': 'yak', 'x': 6.41}]
21 >>> list(sh.items())
22     [('ant', 'hello'), ('cat', {'z': 88 ... 'yak', 'x': 6.41})]
23 >>> sh.close()
```

then later

```

1 >>> sh = shelve.open("shdb", "w")
2 >>> sh["ant"]
3     'hello'
4 >>> sh.close()

```

All of the dictionary methods also work on shelve objects except for the `|` operator, to recap, they are: `%` formatting, `pop`, `get`, `popitem`, `len`, and `update`. But one piece of common coding practice must not be used, and that is an `update` in place of a mutable object.

```

1 >>> sh["horse"] = [ 2, 7, 4, 1 ]

```

Do not do this

```

>>>
1 >>> sh["horse"].append(66)

```

Do it in three steps instead

```

1 >>> x = sh["horse"]
2 >>> x.append(66)
3 >>> sh["horse"] = x

```

There is another alternative called `dbm`. It is used in exactly the same way as `shelve`, but has some restrictions. Essentially `dbm` can only store basic built-in Python types. `Shelve` is in fact `dbm` plus `pickle`, so there isn't much point in using `dbm` alone.

## 28. Exceptions: detecting and handling errors

When you do something wrong, you will normally find that your program stops and an error message is shown. The message includes a listing of all the currently executing functions and at the very end a description of the error. Here are some examples that you'll already be familiar with. Except for the first one, I'll only show the last line of the message.

```

1 >>> def f(x):
2 ...     if x == 1: return math.sqrt(-2)
3 ...     if x == 2: return 1 / 0
4 ...     if x == 3: return math.dennis()
5 ...     if x == 4: return 123
6 ...     print("No match")
7
8 >>> f(1)
9     Traceback (most recent call last):
10      File "<pyshell#36>", line 1, in <module>
11        f(1)
12      File "<pyshell#35>", line 2, in f
13        if x == 1: return math.sqrt(-2)
14     ValueError: math domain error

```

```
15 >>> f(2)
16     ZeroDivisionError: division by zero
17 >>> f(3)
18     AttributeError: module 'math' has no attribute 'dennis'
```

The names `ValueError`, `ZeroDivisionError`, and `AttributeError` are called exceptions.

Even control-Cs can be tamed this way. Control-C causes a `KeyboardInterrupt` which can be handled in exactly the same ways as the errors we are working on now. Capturing control-Cs can be dangerous. You don't want to end up with a program that you can't stop if something goes wrong.

## i. Catching exceptions

Any statement or group of statements in your program can be surrounded by an exception handler that will detect any or all exceptions and let you deal with them however you want. When something goes wrong, an exception is raised, and all parts of your program are terminated as though the functions had hit a `return` statement, until you get back to a matching exception handler. In the examples the program stopped with an error message because it didn't have any exception handlers. But Idle itself has one, and all it does is print the information and continue with Idle's interactive loop. Here is a simple exception handler:

```
1 >>> def user(a):
2 ...     b = 9
3 ...     print("user has started")
4 ...     try:
5 ...         b = f(a)
6 ...         print("Received", b)
7 ...     except ValueError:
8 ...         print("You used an inappropriate value")
9 ...     except AttributeError:
10 ...         print("Bad name")
11 ...         b = -1
12 ...     print("user is ending")
13 ...     return b
14
15 >>> user(1)
16     user has started
17     You used an inappropriate value
18     user is ending
19     9
20
21 >>> user(2)
22     user has started
23     ...
24     ZeroDivisionError: division by zero
25
26 >>> user(3)
27     user has started
28     Bad name
29
```



```

30     user is ending
31     -1
32 >>>
33     user(4)
34     user has started
35     Receieved 123
36     user is ending
37     123

```

Exceptions and their handlers can be used to solve a sometimes annoying problem. The `break` statement only breaks out of a single loop. What if you have nested loops and want to break out of more than one of them?

```

1 >>> class LoopBreaker(Exception):
2 ...     pass
3
4 >>> for a in range(4):
5 ...     try:
6 ...         for b in range(4):
7 ...             for c in range(4):
8 ...                 for d in range(4):
9 ...                     print(a, b, c, d, sep = "", end = " ")
10 ...                     if a + b + c + d > 4:
11 ...                         raise LoopBreaker()
12 ...     except LoopBreaker:
13 ...         pass
14 ...     print(a)
15
16     0000 0001 0002 0003 0010 0011 ... .. 0021 0022 0023 0
17     1000 1001 1002 1003 1010 1011 1012 1013 1
18     2000 2001 2002 2003 2
19     3000 3001 3002 3

```

`try` and `except` may be followed by any number of statements, and each `try` may have any number of `excepts`. The `try` statements are executed as normal. If an exception occurs and we have provided an `except` for it, the statements of the `except` are executed, and everything continues as normal. Notice that the "Received" message only appeared in the case that had no exception. If an exception occurs that we did not provide an `except` for, we will not catch it. Our exception handler will exit just like any other piece of code, and the exception will continue until something does catch it.

The names of exceptions that follow the `except` must all be classes that inherit from `BaseException`. Inheritance is important with `excepts`. An exact match of types is not required. `except Super:` will also catch exceptions of type `Sub` if `Sub` is a subclass of `Super`.

If you want to handle a number of exceptions in exactly the same way, you can write a single `except` that has all of their names in parentheses as in

```

1 ...     except (AttributeError, ValueError):

```

If you want something that will catch every possible exception, just say `except:` on its own. You can think of the whole thing as being like an `if` statement. The `try` is like `if`. The `except xxxs` are like `elifs` and the lone `except:` at the end is the `else`. Just like with an `if` statement, the `excepts` are tried in turn until the first match is found, so a lone `except:` should only be at the end of the list.

An exception handler may also have an `else:` clause and a `finally:` clause. The `else:` statements are executed if no exceptions occurred in the `try:` statements. The `finally:` statements are always executed no matter what, even if an exception occurred and we failed to catch it. In that last case, the `finally:` statements are executed before the exception continues to unwind all of our function calls. Any `else:` must come after all of the `except:s`, and a `finally:` must be last of all.

The following should be taken as an addition to the `user` function. The first and last lines of this little snippet are statements that the function already includes, so you can be sure you're putting these new statements in the right place.

```
1 ...     b = -1
2 ...     else:
3 ...         print("everything went well")
4 ...         finally:
5 ...             print("You can't keep me quiet")
6 ...         print("user is ending")
7
8 >>> c.user(1)                                # an exception we caught
9     user has started
10    You used an inappropriate value
11    You can't keep me quiet                    # the finally
12    user is ending
13    9
14
15 >>> c.user(2)                                # an uncaught exception
16    user has started
17    You can't keep me quiet                    # the finally
18    Traceback (most recent call last):        # the usual fatal error
19    ... ..
20    ZeroDivisionError: division by zero
21
22 >>> c.user(4)                                # no exception at all
23    user has started
24    Receieved 123
25    Everything went well                       # the else
26    You can't keep me quiet                    # the finally
27    user is ending
28    123
```

## ii. Getting information from an exception

Exceptions, like everything else, are objects. The names, such as `ValueError`, are the names of the classes that the exceptions belong to. You can easily capture the

exception object and sometimes find extra useful information inside it. To capture the exception object, just put as followed by a variable name just before the except's colon. The `dir` function is a good way to examine things. I'm going to use a global variable, `x`, to bring the exception object outside the exception handler, and we'll see just which non-existent attribute I tried to access. Many exceptions, such as `ZeroDivisionError`, have no useful extra information to provide, but a lot do.

```
1 >>> x = 0
2 >>> try:
3 ...     math.dennis()
4 ... except (AttributeError, ZeroDivisionError) as e:
5 ...     print(e)
6 ...     x = e
7
8 >>> type(x) == AttributeError
9     True
10 >>> x
11     AttributeError("module 'math' has no attribute 'dennis'")
12 >>> dir(x)
13     [ ... .. 'args', 'name', 'obj', ... .. ]
14 >>> x.name
15     'dennis'
```

If you have a general `except:` to catch all exceptions but still want to see what the exception was, you need a little trick. You can't say `except as e:`, but just remember that superclasses catch subclasses and say `except BaseException as e:` instead.

To really see everything that led up to the exception, the `Exception` object itself contains everything you need. If `ex` is the `Exception` object:

```
ex.__class__.__name__
    is the name of the kind of exception, such as "AttributeError".
ex.args
    is a tuple of all of the information provided when ex was created, something
    like ("module 'math' has no attribute 'dennis'", )
ex.__traceback__
    is a traceback object that represents exactly where in your program the
    exception occurred. Let's suppose that the AttributeError occurred on
    line 6 of D:\python\badprog.py in a function called third. third was
    called from line 9 of that file, inside the function second. second was called
    from line 14 in a function called first, and first was called from line 17,
    not inside any function. Then:
ex.__traceback__.tb_frame.f_code.co_filename
    will be "D:\python\badprog.py"
ex.__traceback__.tb_lineno
    will be 17. Note that this is not necessarily what you might expect. Line 17
    is the first call, not the place where the exception occurred.
ex.__traceback__.tb_frame.f_code.co_name
    will be "<module>" to indicate that we were not in any function then,
```

```

ex.__traceback__.tb_next
    will be another traceback object, this time representing the next call, from
    first to second at line 15. So to keep the lines from getting too long, I'll
    save the next traceback as exn before we look at it. The file name remains
    the same, so I'll ignore it.
exn = ex.__traceback__.tb_next
exn.tb_lineno is 15
exn.tb_frame.f_code.co_name is "first".
exnn = exn.tb_next
exnn.tb_lineno is 11
exnn.tb_frame.f_code.co_name is "second".
exnnn = exnn.tb_next
exnnn.tb_lineno is 7
exnnn.tb_frame.f_code.co_name is "third", and finally
exnnn.tb_next is None, we have reached the point where the exception
happened.

```

After importing `traceback`, the class method `traceback.print_tb(...)` can be given a traceback object such as `ex.__traceback__` as its parameter and it will print everything in human-readable form.

If you catch an exception but then discover that you can't handle it after all, just put a `raise` statement with no operand as one of the `except:` statements. The exact same exception is re-raised, and will continue unwinding all the function calls until it reaches a suitable exception handler.

```

1 >>> try:
2 ...     print(math.herbert)
3 ...     except AttributeError as e:
4 ...         if e.name == "dennis":
5 ...             raise
6 ...         else:
7 ...             # pretend to correct the problem
8 ...             print("Everything is OK now")
9
10     Traceback
11     ... ..
12     AttributeError: module 'math' has no attribute 'herbert'

```

### iii. Raising exceptions

If you want to cause an exception deliberately, you must create an object that inherits from `BaseException` (or `Exception` if you want to save a bit of typing) and put it as the operand of a `raise` statement. You can re-use existing exception classes or create your own. Any parameters you give to the exception's constructor will be available as the object's `args` attribute.

```

1 >>> def f():
2 ...     for i in range(10):
3 ...         if i > 7:

```

```

4 ...     raise ValueError("It got too big")
5 ...     print("done")
6
7 >>> try:
8 ...     a = f()
9 ... except Exception as e:
10 ...     print(type(e))
11 ...     print(e.args)
12
13     <class 'ValueError'>
14     ('It got too big',)
15
16 >>> class TooHotError(Exception):
17 ...     pass
18
19 >>> try:
20 ...     raise TooHotError(95, "Fahrenheit")
21 ... except Exception as e:
22 ...     print(e.args)
23     (95, 'Fahrenheit')

```

The statement “assert condition, string” is really just a conditional exception raiser, it is equivalent to

```

if __debug__:
    if not condition:
        raise AssertionError(string)

```

#### iv. Making a class compatible with the with statement

We'll start with an example of something dangerous that needs to be handled carefully. Suppose we are working in a laboratory that investigates giant deadly laser beams. Software to control those things could be provided in the form of a class whose methods do the work. Here it is, but only showing the declarations, the real stuff has been left out.

```

1 >>> class DeadlyLaserBeam:
2 ...     def __init__(self, unit_name):
3 ...         ...
4 ...     def turn_on(self):
5 ...         ...
6 ...     def turn_off(self):
7 ...         ...
8 ...     def aim(self, alt, az):
9 ...         ...

```

To control a laser, we create a `DeadlyLaserBeam` object, giving the constructor the name of the particular laser machine we want. After that we have the option of turning it on, turning it off, and changing the direction it is aimed in (`alt` and `az` are short for altitude and azimuth, the coordinate system often used for aiming telescopes).

But deadly laser beams are dangerous things. What if an error makes the program crash while the laser is still turned on? Who knows what it might burn a hole through. Being very careful to catch every possible exception would keep things safe, and the `with` statement is just a convenient way of doing exactly that automatically. It might be used like this:

```
1 >>> with DeadlyLaserBeam("7G") as dlb:
2 ...     dlb.turn_on()
3 ...     (alt, az, duration) = calculate_target(1)
4 ...     dlb.aim(alt, az)
5 ...     time.sleep(duration)
6 ...     (alt, az, duration) = calculate_target(2)
7 ...     dlb.aim(alt, az)
8 ...     time.sleep(duration)
```

The problem is of course when `calculate_target` fails. `time.sleep(n)` just makes a program pause, doing nothing, for `n` seconds. It gives the laser beam enough time to do its thing. To make that `with` statement work, we need to add two methods to the class. `__enter__` which doesn't have to do anything more than return `self`, and `__exit__` which should do some work.

```
1 ...     def __enter__(self):
2 ...         return self
3 ...
4 ...     def __exit__(self, etype, eval, etrace):
5 ...         self.turn_off()
6 ...         if etype != None:
7 ...             print("Warning, an", etype, "error occurred")
8 ...         return True
```

This is what the little example does. `with DeadlyLaserBeam("7G") as dlb:`

First, it does this.

```
obj = DeadlyLaserBeam("7G")
```

just evaluating the `with` expression and keeping the result, then

```
obj = obj.__enter__()
```

so the `__enter__` method gets a chance to perform any required set-up, perhaps modify the object, or even replace it with a new one. Then

```
dlb = obj
```

the `as` variable can now be used to access the object. And finally, the statements following the `with` are executed.

While the statements are being executed, all possible errors and exceptions are caught. If an exception occurs, the statements are immediately abandoned, and the `__exit__` method is called. `__exit__`'s parameter `etype` will be set to the class object for the kind of exception that occurred, `eval` will be set to the actual exception object, and `etrace` is a traceback object exactly as we saw a little earlier, you can extract file names, line numbers, and function names from it in exactly the same way.

In short, `__exit__` will always be called no matter what happens. If `__exit__` returns `True`, then the rest of the program after the `with` statements continues as normal.

Many standard types implement `__enter__` and `__exit__` so that they can be used with `with`. The file object returned by `open` is probably the best-known example.

## 29. Dates and times

```
1 >>> import time
2 >>> time.strftime("%X")
3     14:47:21
4 >>> time.strftime("%c")          # other formats later in this section
5     Mon Jul 24 17:41:15 2023
6
7 >>> import datetime
8 >>> t = datetime.datetime.now()
9 >>> t
10    datetime.datetime(2023, 3, 7, 19, 47, 8, 295091)
11 >>> print(t)
12    2023-03-07 19:47:08.295091
13 >>> t.month
14     3
```

The attributes of a `datetime` are `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, and `tzinfo` (time-zone data, not shown because it wasn't set), and there is also a `weekday` method. `microsecond` and `tzinfo` can be ignored.

```
1 >>> t = datetime.datetime(1896, 4, 19, 12, 0, 0)
2 >>> names = ["Mon", "Tue", "Wed", "Thur", "Fri", "Sat", "Sun"]
3 >>> names[t.weekday()]
4     'Sun'
5 >>> datetime.datetime.now() - t
6     datetime.timedelta(days=46342, seconds=28028)
7 >>> datetime.datetime.now() + datetime.timedelta(days = 365)
8     datetime.datetime(2024, 3, 7, 19, 47, 8, 295091)
```

Dates and times are available separately, but there is no equivalent of `now()` for times.

```
1 >>> t = datetime.date.today()
2     datetime.date(2023, 3, 7)
3 >>> t = datetime.datetime(1896, 4, 19, 12, 45, 19)
4 >>> t.date()
5     datetime.date(1896, 4, 19)
6 >>> t.time()
7     datetime.time(12, 45, 19)
8 >>> x = datetime.time(20, 29, 15)
9 >>> x.hour
10    20
```

## ii. Unix-style times

Times and dates are also available in a more primitive but computation-friendly form, using almost the Unix standard of the number of seconds since 1st January 1970, midnight G.M.T. `tm_wday = 0` for Sunday, `tm_yday = 1` for 1st January. Unlike in the original Unix version, `tm_mon = 1` means January and you don't add 1900 to `tm_year`.

```
1 >>> import time
2 >>> t = time.time()
3 >>> t
4     1687906859.8652923
5 >>> time.localtime(t)
6     time.struct_time(tm_year=2023, tm_mon=6, tm_mday=27,
7                       tm_hour=19, tm_min=0, tm_sec=59,
8                       tm_wday=1, tm_yday=178, tm_isdst=1)
9 >>> time.gmtime(t)
10    # the same, except tm_hour=23
11 >>> time.gmtime(0)
12    time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1,
13                      tm_hour=0, tm_min=0, tm_sec=0,
14                      tm_wday=3, tm_yday=1, tm_isdst=0)
```

And there are a lot of other methods too. In particular,

```
1 >>> time.sleep(3.61)
```

will make the program pause its execution, leaving the CPU idle, for 3.61 seconds.

Beware: Control-Cs are not noticed during a sleep under Windows.

## iii. Formatting

The special string formatting operation `f` has codes for extracting information from a `datetime`.

```
1 >>> t = datetime.datetime.now()
2 >>> f"it is {t:%A} {t:%d} {t:%B} {t:%Y}"
3     'it is Tuesday 07 March 2023'
4 >>> "it is {0:%A} {0:%d} {0:%B} {0:%Y}".format(t)
5     'it is Tuesday 07 March 2023'
```

The `strftime` function uses exactly the same format strings. Its first parameter is a format string, and its optional second parameter is a time as returned by `localtime`. If the second parameter is missing, the current local time will be used.

the formats are

`%a:` abbreviated day of week,



%A: full day of week,  
 %b: abbreviated month,  
 %B: full month name,  
 %c: everything, format: Mon Jul 24 17:41:15 2023,  
 %d: zero padded day of month,  
 %-d: day of month, replace - with # under windows,  
 %f: six digit zero padded microsecond,  
 %H: zero padded hour (24-hour clock),  
 %I: zero padded hour (12-hour clock),  
 %-I: hour (12-hour clock), replace - with # under windows,  
 %j: three digit zero padded day of year,  
 %-j: day of year, replace - with # under windows,  
 %m: zero padded month number,  
 %-m: month, replace - with # under windows,  
 %M: zero padded minute,  
 %-M: minute, replace - with # under windows,  
 %p: AM or PM,  
 %S: zero padded second,  
 %-S: second, replace - with # under windows,  
 %U: zero padded week of year if weeks start on Sundays,  
 %w: day of week, 0 = Sunday,  
 %W: zero padded week of the year if weeks start on Mondays,  
 %x: date, format: 07/24/23,  
 %X: time, format: 14:47:21,  
 %y: last two digits of year,  
 %-y: year without century no padding, replace - with # under windows,  
 %Y: full year,

The documentation claims formats %z, %Z, but they just produce empty strings.

#### iv. CPU time

If you want to time a running program, `time.process_time` tells you how many seconds of CPU time have been spent on this program (technically processes) only.

```

1 >>> time.process_time()
2     2.859375
3 >>> time.process_time()
4     2.859375

```

You have probably noticed that the two times are exactly the same, down to the microsecond, and I certainly didn't type that second command in less than a microsecond. That illustrates what CPU time really means: waiting doesn't, only work time is included. Plus the clock isn't really that accurate.

## 30. Minor data structures

Python provides a number of modules that implement data structures beyond those built into the language. They generally wouldn't be difficult to make for yourself, but they do save a bit of effort and debugging.

## i. Queues

The `queue` module provides three useful versions of queues, one of which isn't a queue at all. They are list-like data structures with restricted access. The implementation includes an automatic mutex. That means that if two threads of your program try to update the queue at the same time (they do provide a good means of communication between threads) nothing will go wrong.

The three classes are:

`queue.Queue`, which is a normal queue. You can only add items to the end, and you can only remove items from the beginning.

`queue.LifoQueue` is in fact a stack. You can only add or remove items to or from the end.

`queue.PriorityQueue` is what the name suggests, a priority queue. When things are added, they don't go into any particular position at all. The only thing you can remove is the item with the smallest value. To make this useful, remember that lists and tuples can be compared just as easily as numbers, and so can your own objects if they have an `__lt__` method.

Whichever you want, you'll need to do the equivalent of this:

```
from queue import PriorityQueue
```

The constructors all take one optional parameter, and that is the maximum number of items that the structure can contain. No parameter or parameter equal to zero means there is no limit. If the limit is reached, then attempts to add more items will block (not return) or raise an exception, depending on the context, until an item has been removed.

The three types of queue have the same methods:

`qsize()`, `empty()`, and `full()`: return the number of stored items and `True` or `False` depending on whether the queue is empty or not. They are only useful in a non-threaded program. If you have threads, then `full()` returning `False` is not a guarantee that the queue will still not be full when you execute the next statement.

`put(object, block, maxtime)`, only `object` is required. Adds the object to the queue and that's that, except that if the queue is already full, the other parameters come into play. If `block` is `True` (the default) this method call will wait (i.e. not return) until the queue is not full. If `maxtime` is provided and not equal to `None`, then it is the maximum number of seconds the wait will last. If the maximum time expires or `block` was `False`, the `queue.Full` exception will be raised. `put_nowait(object)` is equivalent to `put(object, False, None)`.

`get(block, maxtime)`, neither parameter is required. If the queue is not empty the one eligible item is removed and returned as the result. If the queue is empty, it is like `put`: if `block` is `True` (the default) then it will wait until the queue is no longer empty. If `maxtime` is provided and not `None`, then it is the maximum number of seconds the wait will last. If the maximum time expires or `block` is `False`, the `queue.Empty` exception will be raised. `get_nowait()` is equivalent to `get(False, None)`.

**BEWARE:** if `put` or `get` is blocking, not even a control-C will stop your program under Windows.

Python's priority queues are of limited use, because adjusting the priority of an item after it has been added doesn't entirely work. It has some effect, but not always the correct one.

## ii. Double ended queues

A double ended queue, or `deque`, is an extension of an ordinary queue. You can both add and remove items at either end. `deque` is part of the `collections` package, so include `from collections import deque`.

The constructor can take no parameters, in which case it creates an empty `deque`, or it can take an iterable, in which case all the objects in the iterable are added, in order, to the end. If you provide an iterable, you may also provide a second optional parameter to specify the maximum number of objects the `deque` can store. The methods are:

`append(object)`, add the object to the end (right) of the `deque`. Returns nothing.

`appendleft(object)`, add the object at the front (left) of the `deque`. Returns nothing.

`pop()`, remove and return the last (rightmost) item.

`popleft()`, remove and return the first (leftmost) item.

`extend(iterable)` and `extendleft(iterable)`, add all the objects from the iterable, in order, at the given end of the `deque`.

`clear()`, remove all objects, make the `deque` empty.

indexing, `deque[i]`, accesses the item at position `i`, leftmost is zero. can be used to look at the item or to change it. Also `del deque[i]` removes an item.

`index(object)`, returns the position of the object in the `deque`, leftmost is zero. It uses `==`, not `is`, to do the comparisons. Raises `ValueError` if not present.

`insert(index, object)`, insert the object so that it appears at the given index. The object that used to be at that position, and all following, move one position to the right.

`remove(object)`, remove the object, moving all later objects one position to the left to fill the hole. `ValueError` if not present.

`rotate(n)`, the last `n` items in the deque become the first `n` items, all other items move `n` positions to the right.

The operator `+` concatenates two deques to create a new one, and `*` by a number produces a new deque equal to that number of copies of the original concatenated together.

### iii. Counters

A `Counter` contains data items, but its primary purpose is to keep a count of the number of times each item has been added. It presents this information in the form of a dictionary. The constructor for a `Counter` takes any iterable the contents of which are added, or nothing at all in which case it starts out empty. Once a `Counter` exists, the `update` method also takes any iterable and adds its contents. Beware, `Counter` is spelled with a capital C.

```
1 >>> from collections import Counter
2 >>> c = Counter()
3 >>> f = open("/home/www/text/barrie/peterpan", "r")
4 >>> for line in f.readlines():
5 ...     c.update(line.split())
6 >>> f.close()
7 >>> c["the"]
8     2169          # "the" is a very popular word
9 >>> c["hippopotamus"]
10    0            # surprisingly no hippopotamusses in Peter Pan
11
12 >>> c = Counter()
13 >>> f = open("/home/www/text/barrie/peterpan", "r")
14 >>> for line in f.readlines():
15 ...     for word in line.split():
16 ...         c.update(word)
17 >>> f.close()
18 >>> c["the"]
19    0            # a string is an iterable, so all of its characters were added
20 >>> c["e"]
21    25213
22 >>> c.most_common(4)
23    [('e', 25213), ('t', 17808), ('a', 14917), ('o', 14068)]
24 >>> c
25    Counter({'e': 25213, 't': 17808, 'a': 14917, 'o': 14068,
26             'h': 13992, 'n': 12762, 'i': 12182, ...
27             ... 'l': 2, 'j': 2, 'z': 1, '^': 1})
28 >>> c.total()
29
```

```
208527      # the sum of all the counts
```

There are four operators that produce new Counters:

```
a + b = everything from both, with their counts added
a | b = everything from both but with maximum count from either
a - b = subtract b's count from a's count, <= 0 means remove
a & b = everything from both but with minimum count from either
```

#### iv. Named tuples

The contents of a normal tuple are accessed with a numerical index. If the position of something within a tuple has some significance, it is critical and difficult to remember the correct indexes. A `namedtuple` replaces numeric indexes with named fields, like attributes in a class. In fact, the constructor for a `namedtuple` doesn't return an ordinary object, instead it returns a new class in which the chosen names really are attributes. The constructor `namedtuple` is a little odd. Its first parameter is the name you want the new class to have, and the second parameter is a space-separated string giving the names of the positions or attributes. The class returned by `namedtuple` is used as a constructor for these new objects.

```
1 >>> from collections import namedtuple
2 >>> nation = namedtuple("nation", "name population area")
3 >>> nepal = nation(area = 147516, name = "Nepal",
4 ...               population = 30666598)
5 >>> spain = nation(name = "Spain", area = 195365,
6 ...               population = 48196693)
7 >>> liechtenstein = nation(name = "Liechtenstein",
8 ...                       population = 38387, area = 62)
9 >>> spain
10 nation(name='Spain', population=48196693, area=195365)
11 >>> nepal.population
12 30666598
```

It is an error if any of the attribute names would be unusable (perhaps something like “while”, “96”, or “\*”), but sometimes the attribute names can not be checked by the programmer, they may have come from the first row of a CSV file for example. In that case, give the keyword parameter `rename = True` to the `namedtuple` constructor, then any bad names will be converted to something safe but meaningless, like “\_2”.

#### v. Chained dictionaries

A `ChainMap` allows you to bring multiple dictionary-like objects together, so that they can all be searched in a single operation. You use a `ChainMap` exactly as you use a dictionary. Searches are applied to each of the included dictionaries in turn, in the order that they were given to the constructor, as soon as one results in a successful search, that is the result. But changes to a value, additions of new

entries, and deletions of entries only happen to the first included dictionary. It is an error (`KeyError`) if you try to delete an entry that isn't in the first dictionary.

```
1 >>> from collections import ChainMap
2 >>> d_one = { "a": 111, "b": 222, "c": 333 }
3 >>> d_two = { "b": 444, "d": 555, "e": 666 }
4 >>> d_three = { "a": 777, "b": 888, "c": 999, "f": "cat" }
5 >>> d = ChainMap(d_one, d_two, d_three)
6 >>> d["a"]
7     111
8 >>> d["b"]
9     222
10 >>> d["e"]
11    666
12 >>> d["f"]
13    'cat'
14 >>> d["b"] = "new b"
15 >>> d["f"] = "new f"
16 >>> d["x"] = 1234
17 >>> del d["c"]
18 >>> d_one
19    {'a': 111, 'b': 'new b', 'f': 'new f', 'x': 1234}
20 >>> d_two
21    {'b': 444, 'd': 555, 'e': 666}
22 >>> d_three
23    {'a': 777, 'b': 888, 'c': 999, 'f': 'cat'}
```

A `ChainMap` has an attribute named `maps`. Its value is a list containing all the included dictionaries in their proper order. You can change the included dictionaries by modifying this list.

The `new_child(newdict)` method adds another dictionary to the front of the list, so that it becomes the first to be searched. If `newdict` is `None` or not provided, a new empty dictionary will be added. This is useful if you want to protect the included dictionaries from change. All changes will only happen to this new first entry.

`parents` appears to be an attribute, but it is a property (a method that you access as though it were an attribute). `cm.parents` is a new `ChainMap` the same as `cm`, but with the first of the included dictionaries removed. It is the opposite of `new_child`.

## 31. Access to operating system features

These things will of course behave differently depending on which operating system your computer has. But as the only likely choices these days are Unix and Windows, the differences should be manageable.

### i. System-independent code

There are a number of standard Python things that don't work under Windows. One way to handle this is to make your module detect what kind of system it is running under and for the things that are system dependent, provide different definitions. Often, on the Unix side, you can just provide a slightly differently named function that directly calls the standard Python function. Then on the Windows side define a function with that same name that performs whatever work-around is needed. This is easy because in Python, imports and function definitions and so on are just normal executable statements, so we can use an `if`.

What should the condition of the `if` be? If you have already found something that behaves differently under the two systems, you could just test that effect. But there are two better ways.

In the `os` module, `os.name` has one of three (currently) possible string values: `'nt'` for Windows, `'posix'` for Unix, and `'java'`. I'm not quite sure why Java would be considered an operating system, but it is possible to run Python from a Java program.

In the `platform` module, there is something rather more specific. `platform.system()` returns `'Windows'`, `'FreeBSD'`, or `'Linux'`. The problem is that those are almost certainly not the only possibilities, and there doesn't seem to be any central place recording them.

So let's suppose we have found two functions, `first` and `second`, that work under Unix but not under Windows:

```
1 >>> import os
2 >>> if os.name == "posix":
3 ...     def x_first(a, b):
4 ...         return first(a, b)
5 ...     def x_second():
6 ...         second()
7 ... else:
8 ...     def x_first(a, b):
9 ...         # whatever is needed to make first work
10 ...     def x_second():
11 ...         # whatever is needed to make second work
```

Now after that conditional part, just make sure you use `x_first` and `x_second` wherever you would normally have used `first` and `second`.

## ii. The `os` module

The `os` module contains a very large number of things indeed. I am only including the few most useful here. If you feel the need for more, the Python documentation is always there. Most of these things are clearly based on standard Unix system calls, so how they behave under windows might be questionable. Generally, if there is a Unix system call for something, it probably has an equivalent in `os`.

`os.getlogin()` returns your user name.

`os.getcwd()` returns as a string your current working directory or folder, and `os.chdir(s)` lets you change it. This will change where python looks for files when you use `open` without specifying an absolute path, but does not alter where `import` looks for modules.

```
1 >>> os.getcwd()
2     'D:\\python'
3 >>> os.chdir("D:\\python\\mcpackage")
4 >>> os.getcwd()
5     'D:\\python\\mcpackage'
```

There are five string values that tell you the way that certain string should be constructed or separated.

	Unix	Windows	
<code>os.pathsep</code>	<code>':'</code>	<code> ';' </code>	separates directories listed in your PATH variable
<code>os.sep</code>	<code> '/' </code>	<code> '\\ ' </code>	separates files and directories in file names
<code>os.pardir</code>	<code> '.. ' </code>	<code> '.. ' </code>	how to refer to the parent directory
<code>os.curdir</code>	<code> '.' </code>	<code> '.' </code>	how to refer to the current directory
<code>os.linesep</code>	<code> '\n ' </code>	<code> '\r\n ' </code>	what appears at the end of an official line of text

These functions provide information about files, the parameter is a file name:

<code>os.path.isfile(s)</code>	is the file an ordinary file that actually exists
<code>os.path.isdir(s)</code>	is the file a directory that actually exists
<code>os.path.exists(s)</code>	is it any sort of file and actually exists
<code>os.path.getsize(s)</code>	what is the size of the file, in bytes
<code>os.access(s, mode)</code>	am I allowed to access this file in this way
	<b>mode can be any bitwise-or (the <code> </code> operator) of <code>R_OK</code>, <code>W_OK</code>, and <code>X_OK</code>. <code>RWX</code> for read write execute. True if you can do <i>all</i> of the <code> </code>'ed together modes of access.</b>

```
1 >>> os.path.isfile("p7.py")
2     True
3 >>> os.path.isdir("p7.py")
4     False
5 >>> os.path.isdir("D:\\Python")
6     True
7 >>> os.path.exists("p7.py")
8     True
9 >>> os.path.getsize("p7.py")
10    2628
11 >>> os.access("p7.py", os.R_OK | os.W_OK)
12    True
```

These functions allow you to manipulate file names and paths:

<code>os.path.split(s)</code>	separate into path and file - a tuple
-------------------------------	---------------------------------------



`os.path.splitext(s)` separate into file and extension - a tuple  
`os.path.abspath(s)` exactly where is this file?  
`os.path.join(s)` connect components of a file's path

```
1 >>> os.path.split("D:\\Python\\p7.py")
2 ('D:\\Python', 'p7.py')
3 >>> os.path.splitext("D:\\Python\\p7.py")
4 ('D:\\Python\\p7', '.py')
5 >>> os.path.abspath("p7.py")
6 'D:\\Python\\p7.py'
7 >>> os.path.join("D:", "Python", "subdir", "hen.py")
8 'D:Python\\subdir\\hen.py'
```

That last one is odd. It never puts the `\\` after a drive letter.

```
os.listdir(path = ".")
```

Returns a list of all the files in the given directory, except for `.` and `..`, and in no particular order. The default value `.` represents the current directory.

```
os.walk(path)
```

Much more helpful than `listdir`. It returns an iterable with one entry for every directory that can be reached from the given path. The first entry will always be for the given path itself. The entries are three-tuples, `[0]` is the name of the directory itself, `[1]` is a list of the names of all subdirectories it contains, and `[2]` is a list of all the other files it contains. For this example, my current directory has a subdirectory called `mcpackage`. `mcpackage` has two of its own subdirectories called `__pycache__` and `junior`, and `junior` has a subdirectory called `littlebits`.

```
1 >>> x = list(os.walk("mcpackage"))
2 >>> len(x)
3 5
4 >>> x[0]
5 ('mcpackage',
6  ['junior', '__pycache__'],
7  ['one.py', 'two.py', '__init__.py'])
8 >>> x[1]
9 ('mcpackage\\junior',
10 ['littlebits', '__pycache__'],
11 ['four.py', 'three.py', '__init__.py'])
12 >>> x[2]
13 ('mcpackage\\junior\\littlebits',
14 [],
15 ['first.txt', 'second.txt'])
16 >>> x[3]
17 ('mcpackage\\junior\\__pycache__',
18 [],
19 ['four.cpython-311.pyc', 'three.cpython-311.pyc',
20  '__init__.cpython-311.pyc'])
21 >>> x[4]
22 ('mcpackage\\__pycache__',
23 [],
24 ['one.cpython-311.pyc', 'two.cpython-311.pyc',
```

```
'__init__.cpython-311.pyc']])
```

```
os.remove(filename)
```

Deletes an ordinary file.

```
os.rmdir(dirname)
```

Deletes a directory, error if it isn't empty.

```
os.mkdir(dirname)
```

Creates a directory, error if it already exists.

```
os.system("command")
```

Opens a new shell and runs the given command in it. Under windows you probably won't see anything unless the command includes some way to cause a pause. A new window will appear, the command will run in it, and it will instantly disappear. Under Unix, the output from the command appears as normal Python output does. In either case, the result returned is the command's exit status.

```
os.popen("command", mode = "r")
```

Is a much better alternative to `os.system`, and almost as easy to use. It runs the operating system command you provide, but returns as its result an object that behaves exactly like a text file to allow interaction with the command while it is running. But it has a serious defect. Unlike with real files, the mode can only be "r" or "w". If it is "r", the command will be run with no standard input. There is nothing that will let you achieve the effect of typing in response to its prompts. Any input it needs will have to be taken from the command line. But it does behave like an ordinary read-only text file, `read` and `readlines` and so on all work as usual.

`oprogram` is a program that prints its name then multiplies the two numbers given on its command line.

```
1 >>> f = os.popen("oprogram 7 9")
2 >>> f.readlines()
3     ['oprogram\n', '7 times 9 is 63\n']
4 >>> f.close()
```

If the mode is `w`, you will be able to use the write method on it to respond to prompts, but all the output it produces will just appear on your screen as normal Python output, and input doesn't work the way you would expect, and output appears in an order that you wouldn't expect either. `iprogram` is a program that prints its name then asks the user to type two numbers, then prints their product, and finally exits with their product as its exit code. Here is a session of me using it outside of Python. To make everything clear, what the computer prints is in the typewriter-like font, what I type is in this text font, italic, and underlined. The symbol `↵` shows exactly where I pressed ENTER. This was done under Unix.

```
$ iprogram ↵
iprogram
Enter a number: 4 ↵
```

```

Enter another number: 5 ✓
The product is 20
$ echo $status ✓
20
$

```

Exactly as you would expect from the description. Now an attempt at the same thing on Idle.

```

1 >>> f = os.popen("iprog", "w") ✓
2 >>> iprog
3     Enter a number: ✓
4 >>> f.write("4\n") ✓
5     2
6 >>> f.write("5\n") ✓
7     2
8 >>> f.close() // 256 ✓
9     Enter another number: The product is 20
10    20

```

The first thing to note is that we got the Python prompt for the next command `>>>` immediately after the call to `popen`, and that was followed by output from the program, I didn't type the `>>> iprog` line. Then after the program said enter a number, I had to press enter to get the Python prompt back. Then, sending the two numbers, 4 and 5, to the program had no immediate effect, all the remaining output came after the call to `close`. This strange ordering is due to the buffering that operating systems apply to data streams. The very last line just saying 20 is the exit code, the result of the division `f.close() // 256`. `close` multiplies the program's real exit code by 256 to produce its result.

`os.environ` contains all your environment variables as a dictionary. You need to be careful in using this, the environment will have different keys depending on the operating system. The first of these examples produces an error under Unix, the second produces an error under Windows.

```

1 >>> os.environ["OS"]
2     'Windows_NT'
    OR
1 >>> os.environ["HOSTTYPE"]
2     'FreeBSD'

```

`os.sys.argv` is a list giving the strings on the command line that started this program. If you are using Idle, it will just contain a single empty string.

`glob.glob(pattern, root_dir = ".")`

Returns a list of all the files found starting from `root_dir` (keyword only) whose names or paths match the `pattern`. `pattern` may contain any of the standard Unix "glob" wild-cards for file name matching, such as `*` and `?`, etc.

```

1 >>> glob.glob("b*.p?")
2     ['bad.py', 'boolean.py', 'brain.py', 'brain2.py']
3 >>> glob.glob("*\\*.py")
4     ['mcpackage\\one.py', 'mcpackage\\two.py',
5      'mcpackage\\__init__.py']
6 >>> glob.glob("*.doc*", root_dir = "D:\\computer")
7     ['bcpl-description.doc', 'bcpl-syntax.doc',
8      'bcpl.doc', 'emulator.doc', 'hardware.docx']

```

If you know about Unix systems programming, `os.fork()`, `os.wait()`, `os.kill()`, and `os.execlp()` and all its cousins do exactly what you would expect.

## 32. Multiple streams of execution - Threads

Most programs, even today, run in a completely linear way. They start at the beginning, run through their code instruction by instruction, and stop at the end. That is the way programming is taught after all. But there are alternatives. If at some point, a program has three tasks to perform, and they are at least mostly independent, why just do them one at a time? Why not speed things up and perform all three tasks at the same time?

That is what threading is all about. A thread is a single linear stream of code execution, and programs can have a lot of threads. Modern computer CPUs are multi-core, which means that they really can be doing many totally different things at the same time. But that isn't necessary. If you have a single core CPU, it can only really be doing one thing at a time, but even then software can make that irrelevant. Time sharing is where one thread of execution is allowed to run for a fraction of a second, then it is suspended and another thread gets a turn. This technique has been around since long before anyone came up with the idea of a thread.

But that is the point? If you're only timesharing, having multiple threads isn't going to speed anything up. The fact is that it still can, but not by much. The really important thing is that multiple threads can significantly simplify software design. Think of a web server or a multi-play game server. It will be dealing with quite a number of connected clients (people playing the game) at the same time, and activity will come from those clients at totally unpredictable times. With normal programming techniques it would have the complex job of repeatedly checking on every connection to see if anything needs to be done yet, queueing up those things that need to be done so they can be handled in a fair order, and making sure that activity on one connection won't leak over and affect what is happening on another. With multi-threading, you can have one thread for every client. All it has to do is constantly wait for activity and immediately deal with it. No chance of any confusion between clients. And it doesn't matter how many cores your CPU has or hasn't got, it still provides a good solution to the problem just by simplifying the design requirements.

Unfortunately, Python has a thing called the Global Interpreter Lock, which means that even if you have multiple cores, only one thread can actually be executing Python code at a time. All the others will be stuck waiting for their turn. This doesn't diminish the design improvements, but it is not good for speed. If that is a concern then you need true multi-processing, which will be covered soon.

Python provides two ways to make use of threads. One is the `threading` module, which is more Python-like, and the other is the `_thread` module which sticks very close to the way Unix systems handle threads.

## i. Threading

Getting started is quite easy. You write functions to perform the tasks that you want executed concurrently. There are no limits on those functions, they can do anything you want. The `threading` module defines a class called `Thread`. The constructor demands keyword parameters only. `target` should be the function that you want to be run. `name` should just be a string, something you can use to tell the threads apart (it is optional, a name will be made up if you don't provide one), and `args` should be a tuple of all the parameters your function needs to be given.

Once you have created all the `Thread` objects, just use each of their `start` methods, no parameters, and it just happens. If you want your main program or function to wait until all your threads have finished, then it should call each of their `join` methods, also no necessary parameters.

The `threading` module provides a `current_thread` method. It returns the `Thread` object for whichever thread is currently running, so when your thread function uses it, it will receive its own thread object. `Thread` objects have a property called `name`, so they can find out their own names in order to make anything they print be attributable to them.

```
1 >>> import threading
2     import time
3
4     def slow(delay, limit):
5 >>>         me = threading.current_thread()
6 >>>         for i in range(1, limit):
7 >>>             print(me.name, i)
8 >>>             time.sleep(delay)
9 >>>             print(me.name, limit, "done")
```

We'll be using that thread function to drive each of three separate threads. They will of course all have different names, delays and limits. If we just run the thread function normally we would see this happening very slowly (depending on the value of `delay`)

```
1 >>> slow(1, 5)
2     MainThread 1
3     MainThread 2
```

```

4     MainThread 3
5     MainThread 4
6     MainThread 5 done

```

But of course, we're going to run them as threads

```

1 >>> def run():
2 ...     one = threading.Thread(target = slow,
3 ...                               name = "one",
4 ...                               args = (2, 20))
5 ...     two = threading.Thread(target = slow,
6 ...                               name = "two",
7 ...                               args = (1, 20))
8 ...     three = threading.Thread(target = slow,
9 ...                               name = "three",
10 ...                              args = (5, 10))
11 ...     one.start()
12 ...     two.start()
13 ...     three.start()
14 ...     one.join()
15 ...     two.join()
16 ...     three.join()
17 ...     print("It's all over")
18
19 >>> run()
20     onetwothree   111
21
22
23     two 2
24     one 2
25     ... ..

```

The jumble at the beginning is because all of the threads started at once, there haven't been any `time.sleeps` yet. Once `print` gets started on an object, it can get it printed quite quickly, but the slices of time each thread is given are very short, so `print` just has enough time to print the first thread's name, `one`, before the second thread gets its first turn, so we see `onetwothree` all run together. The `print` gets enough time to print the spaces that separate `me.name` and `i`, then we see all of their first `i` values, `1`, then we get the three newlines all at once too. After that, the `time.sleep` delays usually stop the threads from talking over the top of each other, but it still can happen. Timing for `sleeps` isn't very accurate. This carries on for a while until the end when we see

```

1     ... ..
2     one 18
3     three 10 done
4     one 19
5     one 20 done
6     It's all over

```

Beware of two important and closely related problems.

1. There is no way to kill a thread. Even the thread that created it has no power over it. The only way a thread will stop is if its function voluntarily finishes or if an unhandled exception occurs.

2. Control-C has no effect on any but the main thread. An uncaught Control-C will kill the main thread, but will not touch the others. When the main thread is killed, its mainness is not inherited by any other thread.

This means that you have to be vary careful. If you need to stop a program that has threads, all you can do is close the window that it is running in. If you're using Idle that means you lose everything you have built up. On a Unix system, and this includes Mac, you can send a -KILL or -HUP signal from another process, but you will still lose everything.

If I remove all three calls to `join`, things would be different:

```
1 >>> run()
2     onetwothreeIt's all over
3     111
4 >>>
5
6     two 2
7     one 2
8     ... ..
9     ... ..
10    one 18
11    three 10 done
12    one 19
13    one 20 done
```

Here, there are four threads competing for CPU time, the original thread that `run` is running in is now one of them, it is no longer waiting in the background. That's why we see `It's all over` right up against the `onetwothree`. Then `run` is finished, so it exits and we get Python's prompt again. I could type more Python expressions and commands while the three remaining threads are still running, and they would be obeyed and have their results displayed immediately. At the end of the sample, we do not see a Python `>>>` prompt because we've already had it. There is nothing special about a thread finishing. If this were a whole "program", `run` because of a double-click on its icon, then the program would not finish until every one of its threads has finished.

There is a slight variation on this behaviour when you create a `dæmon` thread. This is not the normal meaning of `dæmon` in computing, but at least Python spells it correctly. A `dæmon` thread is a minor thread, only there to serve. When the main thread terminates, all `dæmon` threads will also be terminated, the program will not wait for them. A thread is made `dæmonic` in one of three ways:

1. Give the constructor a keyword parameter called `daemon`. `True` means it will be, `False` means it won't be, and `None` means it will be the same as the parent (current running) thread is. `threading.Thread(target = Slow, ..., daemon = True)`
2. Once you have a `Thread` object (returned by the constructor or obtained from `threading.current_thread()`), use its `setDaemon` method, in the current example, that would be `me.setDaemon(True)`. For some reason this second method is now frowned upon by the Python community.
3. Assign to the `Thread` object's `daemon` property: `me.daemon = True`. Or of course `me.daemon = False` to set it back to normal.

You can find out whether a thread is dæmonic or not either by using its `isDaemon` method (also disapproved of) or by looking at the value of its `daemon` property.

Unfortunately, this does not help with the Control-C problem. An uncaught exception will kill any thread except the main one, and it is only the main one that Control-Cs are delivered to. When the main thread has an uncaught exception, any program or function or whatever that it is running does get stopped, but the thread itself stays alive. If it didn't stay alive, there would be nothing left to >>> prompt for the next command or obey it.

All threads share the same global variables, along with all local variables that existed before the thread was created. That means programmers need to very carefully consider whether a global variable really should be global or not. If more than one thread is modifying it at different times, who knows what the result might be? It would be nice if each thread had its own namespace, so that everything it uses can access variables that are global within the thread, but invisible outside.

`threading` provides a special class, `threading.local`, for that. Just create an instance, the constructor has no parameters, `L = threading.local()`, and make it available to all of your threads, perhaps as a global variable, perhaps as a parameter. Every time you look at that object from the same thread, you will see the same thing, but when you look at it from another thread you see something different. `L` is an object that gives direct access to another object which is selected based on which thread is current.

You can create what are called thread local variables just by adding attributes to `L`. `L` has no attributes of its own with names that don't begin with an underline, so you can call those variables anything you want. Unfortunately you can't prime it in advance. You can not create `L`, give it some useful attributes, and then give it to your threads. They will be different threads, so they won't see the same object that you initialised.

A less mysterious method is also available. You can add attributes to any object you like. Each thread has its own `Thread` object, so you could use that as the object that gets your thread local variables as attributes. But that is dangerous. If you choose to give a thread local variable the same name as an existing `Thread` attribute, everything will go wrong.

More `threading` class methods:

`enumerate()`

Returns a list of all the currently active `Thread` objects. Active and alive mean the same thing. A thread is alive from the moment you create it until its function finally exits. It makes no difference whether it is asleep, waiting, or actively doing something. So I'll stick with "alive" rather than "active". There will always be a main thread from which you created all the others, and if you are using `Idle` there is a surprise extra called `SocketThread` which allows communications between `Idle`'s components.



`active_count()`

Returns the number of alive threads, this is the same as the length of `enumerate`'s result.

`main_thread()`

The same as `current_thread` except of course it only gives you access to the main thread's `Thread` object.

`stack_size()`

You need to be careful with this. All programs use a large stack to store local variable and parameter values and other essential information for all currently active function calls. Each thread has its own independent sequence of function calls, so it must have its own stack to keep them in. When called with no parameters, `stack_size` returns the currently selected size for every newly created thread. This is automatically set for every run of Python, you don't have to do it. If `stack_size` is given a parameter, that is used to set the stack size that will be used for all newly created threads. The parameter may be zero, to say "go back to the original default", or any number  $\geq 32768$  for a specific size.

Thread objects have a few more features too:

`native_id`

Once a thread has been started, this property will deliver an `int` value that uniquely identifies this thread. It is supposed to let you create some kind of dictionary of all of your threads, you would obviously need a unique key for something like that. Unfortunately, `native_id` is only guaranteed to be unique while the thread is alive. After that, `native_ids` can be re-used for other threads.

`is_alive()`

What the name suggests. When a thread exits, the `Thread` object that represents it isn't going to just disappear. `is_alive` lets you validate a `Thread` object. It only becomes `True` when the `start` method is called.

`join(timeout)`

We have already seen the `join` method, but this is a variation. If provided, the `timeout` parameter should be a float, and will be the maximum time that will be waited. If the thread being joined is still alive after this period of time, `join` will exit anyway. `join` always returns `None`, you need to call `is_alive` after a `join` with a `timeout` to find out whether the thread really did exit or not.

More about the stopping problem. The problem of not being able to stop a thread can be relieved, but at quite a cost. The idea behind it is very simple. You create a global variable `must_stop`, set to `False` every time the program starts. Then you design all of your thread functions so that they frequently check this variable and stop themselves by raising an uncaught exception if it is `True`. Finally, you modify

your main thread to catch Control-C exceptions and set `must_stop` to `True` when they happen.

One problem here is that Control-C's are not noticed during a `time.sleep()` or a `join()` under Windows, so you must call your `joins` with a reasonably small time-out, and put them in a loop for as long as there are no new threads left alive. What if you wanted your main program to do something useful while the threads are running? Either you can surround the entire remainder of your program with a `try` to catch Control-C, or you just create another thread to do whatever it was that you wanted the main program to do.

The really big problem is that this approach is not compatible with a lot of programming paradigms. Threads don't usually just run in simple loops. For this to work, everything that can cause repetition must include a check on `must_stop`. And there is nothing you can do about functions that you didn't write yourself taking a long time to complete, you can't make them look at `must_stop`.

But here it is anyway. First the modified thread function:

```
1 >>> def slow(delay, limit):
2 ...     global must_stop
3 ...     me = threading.current_thread()
4 ...     name = me.name
5 ...     for i in range(1, limit):
6 ...         print(name, i)
7 ...         time.sleep(delay)
8 ...         if must_stop:
9 ...             raise SystemExit()
10 ...     print(name, limit, "done")
```

Now the trivial stop function:

```
11 >>> must_stop = False
12
13 >>> def stop():
14 ...     global must_stop
15 ...     must_stop = True
```

And finally the modified main program:

```
16 >>> def run():
17 ...     global must_stop
18 ...     must_stop = False
19 ...     one = threading.Thread(target = slow, ...
20 ...     two = threading.Thread(target = slow, ...
21 ...     three = threading.Thread(target = slow, ...
22 ...     original_count = threading.active_count()
23 ...     one.start()
24 ...     two.start()
25 ...     three.start()
26 ...     try:
27 ...         while threading.active_count() > original_count:
28 ...
```

```

29 ...     one.join(0.1)
30 ...     two.join(0.1)
31 ...     three.join(0.1)
32 ...     except KeyboardInterrupt:
33 ...         stop()
           print("It's all over")

```

## ii. Timers

The class `threading.Timer` simplifies a common thread task, making something happen at some point in the future. The constructor takes a few parameters, the first is the delay in seconds, the second is the thread function, and after that you can provide a tuple of the parameters that your thread function will need, just like when creating a `Thread`. Then you call the `Timer` object's `start()` method, and you're done. A `Timer` can be cancelled, by calling its `cancel()` method, but only while it is still waiting for the delay to expire.

Sadly, there is no variation on `Timer` that makes a task happen repeatedly with a given delay between calls. Getting over that deficiency is quite ugly, there is nothing you can do to re-trigger a `Timer` object once it has done its thing. You would have to make your own subclass of `Thread`, and if you take into account the need to be able to stop it eventually, that will not be trivial.

In fact, this is what it takes. Warning: it is not a pretty sight.

```

1 >>> import threading
2 >>> import time
3 >>> import math
4
5 >>> class repeater(threading.Thread):
6 ...
7 ...     class stopper:
8 ...
9 ...         def __init__(self):
10 ...             self.must_stop = False
11 ...
12 ...         def stop(self):
13 ...             self.must_stop = True
14 ...
15 ...         def can_run(self):
16 ...             return not self.must_stop
17 ...
18 ...     sleep_unit = 0.1
19 ...
20 ...     @staticmethod
21 ...     def pause(how_long, signal):
22 ...         (part, wholes) = math.modf(how_long / repeater.sleep_unit)
23 ...         for i in range(int(wholes)):
24 ...             if signal.must_stop:
25 ...                 raise SystemExit()
26 ...             time.sleep(repeater.sleep_unit)
27 ...         if signal.must_stop:
28 ...             raise SystemExit()

```

```

29 ...     time.sleep(part)
30 ...     if signal.must_stop:
31 ...         raise SystemExit()
32 ...
33 ...     @staticmethod
34 ...     def threadfunc(func, init_delay, each_delay,
35 ...                   times, signal, * params):
36 ...         repeater.pause(init_delay, signal)
37 ...         if times == None:
38 ...             while True:
39 ...                 func(* params)
40 ...                 repeater.pause(each_delay, signal)
41 >>>         else:
42 >>>             for i in range(times - 1):
43 >>>                 func(* params)
44 >>>                 repeater.pause(each_delay, signal)
45 ...                 func(* params)
46 ...
47 ...     def __init__(self, func, init_delay, each_delay,
...                   * params, times = None):
...         self.signal = self.stopper()
...         all = (func, init_delay, each_delay, times, self.signal) +
...               params
...         super().__init__(target = repeater.threadfunc,
...                          name = "repeater",
...                          args = all)
...         super().start()

...     def join(self):
...         while True:
...             super().join(0.1)
...             if not super().is_alive():
48                 break

...     def stop(self):
...         self.signal.stop()

def thing_to_do(s):
    print(s)

def run():
    r = repeater(thing_to_do, 0.5, 3.45, "Boo!", times = 4)
    try:
        r.join()
    except KeyboardInterrupt:
        r.stop()
    print("It's all over")

```

The parameters to repeater's constructor are:

- 1: The function to repeatedly call
  - 2: The number of seconds to wait before the first call
  - 3: The number of seconds to wait between subsequent calls
  - 4 and later: The parameters to give to the repeated function each time
- keyword times: The number of times to repeat. None means for ever.

Control-C will stop everything. In many cases it will be preferable in `run` to just say `except:` without naming a particular exception. This is because if something goes wrong while the program is running, and an unexpected exception is raised, you probably still want the thread to stop. If you don't do that, all is not lost. say `x = threading.enumerate()` and print `x`. Identify the one called "repeater", and say `x[i].stop()` with its index.

### iii. Race conditions

Most things a Python program does are not just single actions. Consider a simple update like `x += y * 7.3`. It will happen in three steps, more if you really want to be picky. First, the value of `x` needs to be retrieved from memory. Second, that value has `y * 7.3` added to it. Third, the result gets stored back in memory as the new value for `x`. In normal programming, that doesn't matter at all. With threads, it becomes as serious problem.

Suppose two threads are updating the same variable at almost the same time, lets say thread A is about to do `x += 6` and thread B is about to do `x += 3`, and the initial value of `x` is 10. One possible ordering is that A retrieves the value 10 from `x`, then B retrieves the value 10 from `x`, then A adds 6 to its internal 10 getting 16, then thread B adds 3 to its internal 10 getting 13. Then A stores its 16 as the new value of `x`, and finally B stores its 13 as the new value of `x`, making the final value of `x` be 13. `x += 6` and `x += 3` have both been done, but `x` only increased by 3. With large data structures there are even larger opportunities for disaster. This is called a race condition.

There are a number of ways of avoiding this sort of problem, and all of them are based on the same thing, a mutex. A mutex is just a specially managed flag that says whether it is safe to proceed or not. A thread that is about to change a shared resource (it isn't just simple variables that suffer from this problem, the word resource is usually used to be more encompassing) will wait until its associated mutex says *safe*, then set it to *unsafe*. It can then change the resource comfortable in the knowledge that no other thread that is following the rules can get in. When the operation is complete, the thread sets the mutex back to *safe*. The mutex can not be a simple *True/False* variable because that act of waiting for *safe* then setting to *unsafe* has its own race condition, so mutexes are special objects that can only be accessed through methods carefully designed to avoid all such problems.

Python calls its mutexes `Locks`, `Lock` is a class imported from `threading`. `Locks` are created in the *safe* or *unlocked* position, the constructor has no parameters. There are only two useful methods. `acquire()` means wait until the lock is *safe* then set it to *unsafe* and continue. `release()` means set it back to *safe*. A thread must not call `release` on a lock that it didn't successfully acquire, although Python does not check this. Calling `acquire` on a lock that you have already acquired is a disaster, Python does not detect that you have already got the lock, so the second `acquire`'s wait never ends. `acquire` is one of the things that Control-C's can't penetrate under Windows.

After this setup

```
1 >>> shared_global = 10
2 >>> from threading import Lock
3 >>> shared_globals_mutex = Lock()
```

this is all it takes to do a safe update inside a thread:

```
4 >>> shared_globals_mutex.acquire()
5 >>> shared_global = math.log(shared_global) + math.pi
6 >>> shared_globals_mutex.release()
```

In fact, it's even easier than that. Locks support the `__enter__` and `__exit__` methods that make `with` statements work:

```
7 >>> with shared_globals_mutex:
8 ...     shared_global = math.log(shared_global) + math.pi
```

This is by far the best way to do it. It makes it impossible to forget to do the release operation, which would be a disaster for the whole program.

You do not always have a mutex for every single shared resource, they are usually grouped together with one mutex for a whole group of related resources.

Sometimes you will have a very busy thread, it won't want to just sit around waiting to acquire a `Lock`. If the `Lock` is *unsafe* the thread would prefer to get on with something else and try again later. `Lock` has a useless method called `locked`, it returns `True` if the mutex is *unsafe*. Do not use it. There is a race condition between finding that `locked()` returned `False` and calling the `acquire` method. Instead, make use of `acquire`'s two optional parameters, `blocking` and `timeout`. `timeout` specifies the maximum amount of time, in seconds, that the thread is willing to wait. If the timeout expires, `acquire` just gives up and returns. The default value of `-1` means you can wait for ever. `blocking`, default `True`, means that you are willing to wait at all. setting `blocking` to `False` is the same as setting `timeout` to `0`, it will never wait. Of course, if you are using either of these options, you won't know if you have successfully acquired the mutex when `acquire` returns. Fortunately `acquire` tells you. It returns `True` if you were successful and now control the mutex, and `False` if you need to do something else and try again later.

There are a number of other mechanisms that build on the idea of a mutex.

An `RLock` is exactly the same as a `Lock` except that multiple acquires when you already control the `Lock` are not a problem. But still, acquires and releases must come in pairs. If you do three acquires on an `RLock`, it will not be *safe* again until you have done three releases. This is useful because part of operating on a shared resource might involve another operation on another shared resource. If both of those operations were protected by the same mutex, nothing would ever work.

A Semaphore is like a non-binary Lock. Instead of just being *safe* or *unsafe*, it has a counter. So long as the counter is greater than zero it is considered *safe*. Equal to zero means it is *unsafe*. It should not be able to ever become negative. Semaphore is also imported from `threading`.

Semaphore's constructor has one optional parameter, default 1, it specifies the initial value of the counter.

The `acquire` method, with the same blocking and timeout parameters as a Lock, waits until the counter is greater than zero, then decrements that counter and continues.

The `release` method also has an optional parameter, default 1. It just increments the counter by the value of the parameter and continues.

There is also a `BoundedSemaphore`, which is exactly the same as a Semaphore, except that the initial value given to the constructor is also the maximum value. If a release ever results in the counter exceeding that initial value, a `ValueError` exception is raised.

It should be remembered that the `queue.Queue` class uses a mutex. That makes it into a very useful thing for communications between threads. If two threads need to communicate, make a `Queue` for them. Whatever one thread puts to the `Queue`, the other can get. The `put` and `get` do not require the threads to synchronise, `Queues` hold their contents as long as necessary. The mutex means that even if the two threads do by chance use the `Queue` at the same time, it will still work properly.

An `Event`, also imported from `threading`, represents something that can happen that a thread might want to wait for. It is much less satisfactory than a `Queue`, but does have its benefits. An `Event` object includes a `bool` variable which is initially `False`. Any thread that notices that something interesting has happened can use the `set` method to make it `True`, or if the condition goes away, use the `clear` method to set it back to `False`. Any number of threads can call the `Event`'s `wait` method. They will all wait until the `Event` is set, at which point they will all wake up and continue. `wait` also has an optional `timeout` parameter exactly the same as a `Lock`, but no `blocking` parameter. If you specify a `timeout`, you can still tell what happened. `wait` returns `True` if the `Event` became (or was already) set, or `False` if the `timeout` expired.

Seemingly the least useful of these things is the `Barrier`, also imported from `threading`. A `Barrier` is a way to delay a number of threads until they are all ready at the same time, then let them all continue simultaneously so they can work on something that needs them to be synchronised. `Barrier`'s constructor has at least one and up to three parameters. The first is the number of threads that need to be synchronised. The second, called `action` with default `None`,

should be a parameterless function. The third, called `timeout` with default `None`, is as its name suggests.

Whenever one of the threads involved is ready to synchronise with the others, it should call the `Barrier`'s `wait` method. That will make it wait until the entire expected number of threads (from the constructor) are also waiting. Once they are all waiting, the `action` will be called, and all of the waiting threads will be allowed to continue. Each thread receives a guaranteed-to-be-different `int` in the range 0 to `number-of-threads-minus-one` as the result of `wait`. That means that if you need say three of them to do one thing, and three of them to do another, you can just have an `if r < 3:` based on the result from `wait`.

The `wait` method has an optional parameter also called `timeout`. If the `Barrier`'s constructor and the `wait` method both receive a `timeout` parameter, then the one given to `wait` will be used in preference. If any `timeout` is provided and any thread is left waiting longer than that `timeout`, everything fails. All the waiting threads are woken up and given a `BrokenBarrierError` exception, and the `Barrier` itself is said to be *broken*. If a `Barrier` is *broken*, any future `wait` on it will get an immediate `BrokenBarrierError`. `Barriers` may be used any number of times unless they are *broken*.

A `Barrier`'s `reset` method will set it back to un*broken* with a count of zero waiting threads. The `abort` method can be used to deliberately make the `Barrier` *broken*. If any threads were waiting when either of these methods is called, they will all receive a `BrokenBarrierError` exception. The `Barrier`'s `broken` attribute tells you `True` or `False` whether or not the `Barrier` is *broken*. The `parties` attribute returns the number of threads that need to be waiting before they can proceed, and the `n_waiting` attribute returns the number that are currently waiting.

There is an older, much more primitive module called `_thread`. There isn't much point to it any more, so that is all I will say about it.

### 33. Multiple streams of execution - Processes

This is a much more heavyweight approach to multiple streams of execution. A thread doesn't take up very much in the way of resources, so you can have a very large number of them. A process is a much bigger thing, it is how the operating system keeps together everything required for a whole running program. A process may have a large number of threads, but a thread can't have any processes of its own. Threads can create processes, but they are subprocesses of the whole calling process, they do not belong to the creating thread in any way. Operating systems limit the total number of processes that can be in existence at any time, as well as the maximum number of processes any one user can have. The limits are not usually low these days, but you won't be able to have as many processes as you could have threads.



Having said that, processes are much more capable things, and unlike threads, you can always kill them if you want them to stop. Processes take longer to start, find it harder to communicate with each other, and consume more resources. The Global Interpreter Lock which prevents more than one thread from really executing Python code at the same time does not apply across real processes. Processes do not share any memory (unless you make that happen deliberately), so normal things like global variables and `queue.Queue`s can not be used for communications.

Big Thing: Everything to do with Processes works perfectly and exactly as expected under Unix. Windows is different. The only way I have found to make anything to do with Process work requires that the whole “program” be put in a `.py` file and imported. Then you can run its functions or do whatever you want. If you type the exact same thing into Idle, it will not work, and you will get no error messages.

This is all because Windows processes are strange and complicated things. Under Unix, a new process inherits access to just about everything its parent has, including memory. So whatever functions and things you had set up in Python before creating the new process was still there for the new process to use. The same applies to standard input and standard output. Under Windows, new processes are given a program to run, and they run it from scratch. The program it is given is of course the Python interpreter, but it is started again from nothing. There is nothing practical that can be done about this problem.

The `multiprocessing` module provides a class called `Process`. Its constructor is the same as that of `Thread`, and `Process` objects' `start`, `join`, and `is_alive` methods and the `name` and `daemon` attributes are almost identical too. But a daemon process is not the same as a daemon thread. When a daemon process terminates, it does its best to terminate any and all daemon subprocesses that it created, and daemon processes can not create other processes themselves.

A trivial example:

```
1 >>> import multiprocessing as m
2 >>> import time
3
4 >>> def sleepy():
5 ...     print("start")
6 ...     time.sleep(4)
7 ...     print("stop")
8
9 >>> def run():
10 ...     p = m.Process(target = sleepy)
11 ...     p.start()
12 ...     print("Wait")
13 ...     p.join()
14 ...     return p
```

If all that is in a file called `mp.py`, this happens under Windows:

```

1 >>> import mp
2 >>> x = mp.run()
3     Wait
4 >>> x.exitcode          # there is a four second wait for this prompt.
5     0

```

We didn't see `start` and `stop` because they were printed by a different process which wasn't connected to our standard output.

The `exitcode` attribute is `None` if the process hasn't been started yet or is still running. It is `0` if the process terminated naturally. It is `1` if the process stopped because of an uncaught exception. It is a positive  $> 0$  number  $N$  if the program terminated itself with a call to `sys.exit(N)` which is traditionally done to signal some kind of failure. If it is the negative number  $N$  then the process was killed by Unix-like signal number `abs(N)`. So in summary, the little example above had a successful exit.

On the other hand, if you type exactly the same thing into Idle under Windows, for the reasons given above this happens:

```

1 >>> a = run()
2     Wait
3 >>> a.exitcode          # there was no wait at all for this prompt
4     1

```

And under Unix, exactly the same thing has this as its result.

```

1 >>> a = run()
2     Wait
3     start
4     stop
5 >>> a.exitcode          # there is a four second wait again for this prompt.
6     0

```

There is also a `pid` attribute which is the process' unique identification number assigned by the operating system. It is only valid while the process is running. There are two methods for stopping a process. `terminate` and `kill`, neither have any parameters. They do their best to behave as though a Unix `kill -TERM` or `kill -KILL` command sent a signal to the process. `TERM` (`terminate`) will normally stop a process instantly, but processes that are doing something critical are capable of ignoring it. `KILL` (`kill`) always stops a process instantly and can not be ignored. If the killed or terminated process had control of a mutex or semaphore or anything of that kind, it will never be released. Some resources, such as open files are properly closed. When you have completely finished with a `Process` object and it has stopped, its `close` method will release all remaining resources associated with it.

`multiprocessing.active_children()` produces a list of the `Process` objects for all of the current process' started and not yet ended sub-processes. `current_process()` and `parent_process()` return the expected `Process`

objects, but `parent_process` does not look at the whole Unix process tree, `parent_process` called from the main Python thread will return `None`.

The `multiprocessing` module provides two classes that enable processes to communicate with each other. `multiprocessing.Queue` behaves exactly like `queue.Queue` apart from its ability to connect two processes. Any process that has access to a queue can put things into it and get things out of it. This is a very general approach, a `multiprocessing.Queue` can transmit anything that can be pickled.

A `multiprocessing.Pipe` is intended to be used just by a single pair of processes for private communications. A `Pipe` has two “ends”, one for each process. If two processes use the same end at the same time, the whole thing can end up corrupted.

Pipes are made by calls to the `multiprocessing.Pipe` function (it isn't really a constructor). It returns a tuple of two things which are the connection objects that represent each of its ends. Normally `Pipes` are bidirectional, you can read and write at both ends. If `Pipe`'s optional parameter is `False` it will be unidirectional. `(end0, end1) = Pipe(False)` results in `end0` being read only and `end1` being write only.

Whether using `Queues` or `Pipes`, the best way to make them available to a process is to add them to the process function's parameter list when calling the constructor for `Process`.

`multiprocessing.connection.Connection` is the class for the communication objects returned by `Pipe`. Their important methods are:

`send(obj)` - send the object to the other end of the pipe  
`obj = recv()` - receive a previously sent object

Synchronisation is not needed, and pipes have some storage capacity. A few objects can be sent before the first is received, and `recv` will wait if nothing has been sent yet. If the pipe is closed, `recv` can still receive any objects still in it, but a `recv` on an empty closed pipe raises `EOFError`. `send` and `recv` use the `Pickle` module to make objects transferrable, only picklable objects may be sent.

`send_bytes(obj)` - send the object to the other end of the pipe  
`obj = recv_bytes()` - receive a previously sent object and return it  
`obj = recv_bytes_into(b)` - receive a previously sent object and store it in `b`

These methods do not use `Pickle`, but do require you to convert everything to and from a stream of bytes yourself. `send_bytes` accepts `bytes`, `bytearrays`, and other objects that behave like them. `recv_bytes` always delivers `bytes` objects. Sending a really big object (many megabytes) can make everything fail.

`recv_bytes` has an optional parameter, being the maximum number of bytes it is willing to receive. If the sent object is longer than that, it will not be split. `OSError` is raised and the pipe becomes irreparably broken.

`send_bytes` has two extra optional parameters. The first is called `offset`, and specifies where to start sending from. The first `offset` bytes of `obj` will be skipped. The second, `size`, is the maximum number of bytes to be sent.

For `recv_bytes_into`, `b` must be something like a bytearray. Instead of returning the received object, it is stored in `b`. There is an optional second `offset` parameter which says where in `b` to start storing it. This allows multiple objects to be appended into the same buffer. It returns the number of bytes received.

`close()` - close the pipe

`poll()` - return `True` if there is any data ready to be received. It has an optional `timeout` parameter: if there is no data ready, it will wait for up to that many seconds before returning.

`multiprocessing` provides its own versions of `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, `Event`, and `Barrier`. They work in the same ways as the threading versions, but they are less frequently used. The things most commonly used for inter-process communications have their own built-in mutexes.

If you really want to go down to the lowest levels, you can always use the `os.fork()`, `os.execlp()`, and related functions.

## v. Shared Memory

Shared memory is only relevant to multiprocessing. Threads share memory already, so have no need of it.

Memory shared between two processes uses the `multiprocessing` `Value` or `Array` classes, and they completely depend on types understood by the C language. Other Python objects can not be shared. There is no inescapable reason for that not being possible, but it seems as though no module supporting it is available. These are the accepted `ctypes`.

letter	ctypes	Python type	usual size
	<code>c_bool</code>	<code>bool</code>	1
b	<code>c_byte</code>	<code>int</code>	1
	<code>c_char</code>	<code>bytes, len = 1</code>	4
	<code>c_char_p</code>	<code>bytes, N.T.</code>	
d	<code>c_double</code>	<code>float</code>	8
f	<code>c_float</code>	<code>float</code>	4
i	<code>c_int</code>	<code>int</code>	4
l	<code>c_long</code>	<code>int</code>	4
	<code>c_longdouble</code>	<code>float</code>	8

q	c_longlong	int	8
h	c_short (signed)	int	2
H	c_short (unsigned)	int	2
	c_short	int	2
	c_size_t	int	8
	c_ssize_t	int	8
B	c_ubyte	int	1
I	c_uint	int	4
L	c_ulong	int	4
Q	c_ulonglong	int	8
	c_ushort	int	2
u	c_wchar	str, len = 1	4
	c_wchar_p	str	
	c_void_p	int	8

In that table, names like `c_int` and `c_short` refer to the plain signed C types `int` and `short` int. Those with the letter `u` after the underline, like `c_uint` and `c_ushort` refer to the unsigned versions `unsigned int` and `unsigned short int`.

`Value(type, initv, lock = True)`

Creates a shared variable object of the given type and with the given initial value.

`type` may be a single character string equal to one of the letters in the above table, or one of the `ctypes` names. Unlike with normal Python variables, the type is enforced: if you use `'i'`, you can only store an `int`. floats and strings will cause `TypeError` exceptions.

`lock` may only be provided as a keyword parameter. If `lock` is `False`, no mutex will be used for accesses to this object, it will be unprotected against race conditions. If `lock` is `True`, a new `RLock` object will be created just for this shared variable. If `lock` is a `Lock` or `RLock` object, it will be used for all accesses to this shared variable.

`Value` objects have a `value` attribute which may be used to see or to change the value they store. They also have a `get_lock()` method to access their mutexes.

The description of the `lock` parameter is likely to lead to unsafe code. A `Value`'s mutex is only acquired for the duration of a single Python operation. Something like `v.value *= 6` is not a single operation. Remember that even when an object has an `__imul__` method, the `obj *= 6` operation takes two steps. First `__imul__` is called on `obj` and `6`, and it usually modifies `obj` in place, which would be good, but then it returns `obj` (or possibly even a totally new object) as its result, and as a second step, that result is assigned as the new value for `obj`. For updates, you must explicitly deal with the mutex yourself, preferably with a `with` statement.

```
1 >>> v = multiprocessing.Value('i', 44)
2
3 >>> with v.get_lock():
4 ...     v.value *= 6
```



```
20     thur) Building\n<tr><td colspan=5>&nbsp;    \n<tr><td>E-mail:<td><t  
... ..
```

If you need more information, the along with (or instead of) the `r.read()`, you can look at `r.status` and `r.getheaders()`:

```
21 >>> r.status  
22     200  
23 >>> r.getheaders()  
24     [('Date', 'Sun, 30 Jul 2023 23:14:25 GMT'),  
25      ('Server', 'Apache/2.4.56 (FreeBSD)'),  
26     ... ..
```

The status is an indicator of success or failure, always an int. Anything between 200 and 299 indicates success. Everybody knows about the 404 Not Found response. The headers are the extra bits of information provided by the web server to tell you more about the file you received. They nearly always include a Last-Modified telling you the age of the file and a Content-Length telling you its size in bytes.

The value returned by `urlopen` is an `http.client.HTTPResponse` object, which is covered more fully in the clients for the HTTP protocol subsection.

If you get involved in this sort of thing, `urllib` has a lot more to it. It would be worth leafing through the documentation, I can't put everything here.

## ii. Clients for the HTTP protocol

The `http` package provides classes and methods specific to the `www` protocol HTTP. They automate the creation of web servers and clients. It has two essential packages called `client` and `server`, and another two called `cookies` and `cookiejar` for handling those things. I will say nothing more about `cookie` or `cookiejar`.

`http.client` defines an `HTTPConnection` class. This automates getting information from a web server in just a few steps. Create an `HTTPConnection` object to connect to the server. The constructor accepts a single string in the format "hostname" or "hostname:port", or a separate "hostname" string and `port` int, and an optional keyword `timeout` parameter. The port number defaults to 80. There is also an `HTTPSConnection` class, note the extra `S`. It is used in exactly the same way, but uses a secure connection with SSL.

Next, use that object's `request` method. That takes three or four important parameters.

1. The first is the method, a string. "GET" is the most common, it results in you being sent an entire file, with headers that describe it coming first. "HEAD" is the same but you don't get the content, just the headers. "POST" is used when you need to send some data to the server. "GET" and "POST" can both deliver data to the server. With "GET", it is added to the URL most

often after a "?". You often see that when doing a search. "GET" only allows a little bit of data to be sent because there is a maximum length for URLs. "POST" does not modify the URL. Instead, you provide the data to be sent as the body of the request. There are other methods too.

2. The next parameter, also a string, is the URL, the path and name for the file you want. Something like "broccoli/prices.html".
3. The next parameter, `body`, is optional and defaults to `None`. It is the body of the request, the data supplied with a "POST" request. It may be a string, a bytes, or even an open file object.
4. The next parameter, `headers`, is also optional, it is a dictionary, default empty. It specifies extra information to give more detail about the request. It might say who you are, or the type of file you would like to receive. The parameter is not usually provided, but it would look something like

```
{ "Accept": "text/*", "Accept-Language": "en-US" }
```

Once the request has been made, the `getresponse` parameterless method will return an `HTTPResponse` object. Only a few of `HTTPResponse`'s methods and attributes are particularly useful:

`r.status`

The status of the request, as delivered on the first line of the response, 200 means good.

`r.reason`

The string associated with the status, "OK" for 200.

`r.closed`

True or False, has the connection been closed by the server? False should mean that you can send another request on the same `HTTPConnection`, but you can't rely on that. A server can close the connection whenever it wants, and they are never willing to wait a long time.

`r.getheaders()`

A list of two-tuples giving all the metadata that was provided. It usually contains things like the time and what kind of server it is, but also more useful information, like the type of information and its exact length.

`r.getheader(name, default)`

The value of the one header with the given name. If there is no such header, then `default` is returned. `default` defaults to `None`. The header name is not case sensitive.

`r.read(N)`

A bytes object containing the next `N` bytes of the real (data only) response. The empty object `b""` is returned if there is no more data to come.

`r.readinto(buffer)`

`buffer` should be something very much like a `bytesarray`. It is like `read(N)` where `N` is `len(buffer)` except of course the data goes into the



buffer rather than being returned. It returns the number of bytes actually read. `readinto` will not change the size of the bytearray, so when the last `readinto` doesn't manage to fill the bytearray you will need to be careful not to process the left-over bytes from the previous `readinto`.

```
1 >>> import http
2 >>> import http.client
3 >>> conn = http.client.HTTPConnection("rabbit.eng.miami.edu")
4 >>> conn.request("GET", "/findme.html")
5 >>> r = conn.getresponse()
6 >>> r.status
7
8 >>> 200
9 >>> r.reason
10 >>> 'OK'
11 >>> r.closed
12 >>> False
13 >>> int(r.getheader("content-length"))
14 >>> 1264
15 >>> r.getheaders()
16 >>> [('Date', 'Sat, 15 Jul 2023 20:18:03 GMT'),
17 >>>  ('Server', 'Apache/2.4.56 (FreeBSD)'),
18 >>>  ('Last-Modified', 'Fri, 09 Jun 2023 19:23:22 GMT'),
19 >>>  ... ..]
20 >>> ('Content-Length', '1264'),
21 >>> ('Content-Type', 'text/html')]
22 >>> r.read(512)
23 >>> b"<html><head><title>How to find ... by emai"
24 >>> r.read(512)
25 >>> b"l first.<br>\n Don't waste time ... MCA 202 A\n<tr"
26 >>> r.read(512)
27 >>> b'><td><td>Thursday<td> ... </body></html>\n\n'
28 >>> r.read(512)
29 >>> b''
    conn.close()
```

If the file you asked for is not too long, and remember you can look at the `Content-Length` header to find out, you can read it all at once. As soon as the `r = conn.getresponse()` is done, `r.readlines()` will give you the entire contents of the file as an array of bytes objects, one per line. Or `r.read()` will give you the entire contents of the file as a single bytes object. Either way, the bytes objects will include `\n` characters to mark the ends of lines.

Alternatively, you can take more control and send the request and the headers individually:

After creating the connection you can use the `putrequest` method to say which resource you want and what you want to do with it. It takes two string parameters: the HTTP operation, "GET", or "POST", or whatever, and the file or resource name.

Then you use the `putheader` method as many times as necessary for all the headers you want. The call again takes two string parameters: header name and value, e.g.

```
conn.putheader("Accept", "text/*")
```

Then the `endheaders` method triggers the server to do what you asked. In the most common case, "GET", you will be sent the contents of the requested file along with the usual metadata. `endheaders` has an optional `message_body` parameter, which is used to supply the data needed for the "POST" method.

You then use `getresponse` as above to handle the reply.

### iii. Servers for the HTTP protocol

The `http` package also provides a `server` module. It contains two classes for implementing web servers. `HTTPServer` does the basic business of setting up sockets and sending and receiving. `BaseHTTPRequestHandler` actually deals with the GET, PUT and whatever other requests may be received. Except that it doesn't. It provides the basic machinery to enable that, but you have to subclass it to provide the necessary methods.

You would probably be better off using the newer `ThreadingHTTPServer` instead of `HTTPServer`. It can handle the strange way some clients deal with their sockets.

We will create a server that pays almost no attention to what the client requests, but just replies with proper HTML saying what the time is. If the request is for `/index.html`, that is exactly what it does. Any other request is rejected.

We need to Create a server class as a subclass of `BaseHTTPRequestHandler`. In order to handle GET requests, it needs to be given a `do_GET` method. You would need a `do_POST` method to deal with POST requests, and so on for all the other kinds of request.

We have inherited some useful methods and attributes:

`client_address`

a tuple indicating who sent the request, (IP address, port number)

`requestline`

the entire first line of the request, starting with the GET

`command`

the request type "GET", this lets you have a single method to handle multiple types.

`path`

the middle part of `requestline`, the resource/filename that was requested

`headers`

an object that contains all of the headers that were sent with the request. It has a `keys()` method that returns a list of all the header names, and a

`get(key, default)` method that retrieves the value part of a particular header.

`send_response(N)`  
generates the first line of the reply, N is the status, e.g. 200 for OK.

`send_header(name, value)`  
used as many times as needed to produce all the metadata headers.

`end_headers()`  
when all the headers have been sent and we're ready to make the real reply.

`wfile`  
acts like a file open for binary and write only. Whatever you write to it gets sent to the client. When everything is done, just let `do_GET` exit.

`rfile`  
will similarly give you the request body for POST requests etc.

`send_error(code, message, expl)`  
produces the whole reply when there is something wrong with the request. You don't have to use `send_response` or `wfile` or anything else. `code` is the official response like 200 for good, 400 for bad request, 404 for not found, etc. `message` is the text version of that, "not found", etc., and `expl` is any extra text you want to provide. A browser should show the user all three of these things.

`log_request()`, `log_error()`, and `log_message()`  
The default set up is that every request and every use of `send_error` will result in a log message being displayed on your screen. Override these methods to change the format or prevent anything from being printed at all.

The following example reports everything about every request to the user. If the request is for `"/index.html"` the response will tell the time. Any other request is rejected. It will override `log_request`, so that normal requests are not logged at all, and `log_error` so that more of a fuss is displayed. You generally don't override `log_message`, that is just the generic method that the other two call to do the work.

```

1 >>> import http
2 >>> import http.server
3 >>> from http.server import HTTPServer
4 >>> from http.server import BaseHTTPRequestHandler
5 >>> import datetime as dt
6
7 >>> class serverclass(BaseHTTPRequestHandler):
8 ...
9 ...
10 ... # reply() is just a utility to save typing later.
11 ...     def reply(self, s):
12 ...         self.wfile.write(bytes(s, "utf8"))
13 ...
14 ... # change the way things are logged.
15 ...     def log_request(self, * all):
16 ...         pass
17 ...
18 ...     def log_error(self, * all):
19 ...         print("-----\nan error was logged\n- ...
20 ...

```

```

21 ...     super().log_error(* all)
22 ...
23 ... # report every detail of the request.
24 ...     def say_everything(self):
25 ...         print("connection from", self.client_address)
26 ...         print("requestline is", self.requestline)
27 ...         print("command", self.command)
28 ...         print("path", self.path)
29 ...         hds = self.headers
30 ...         for k in hds.keys():
31 ...             print("header", k, hds.get(k))
32 ...
33 ...
34 ... # do_GET is called separately for every single GET request.
35 ...     def do_GET(self):
36 ...         self.say_everything()
37 ...         if self.path != "/index.html":
38 ...             self.send_error(400,
39 ...                             "not allowed",
40 ...                             "you requested " + self.path)
41 ...
42 ...         return
43 ... # the preparation for a good response
44 ...     self.send_response(200)
45 ...     self.send_header("Server", "LittlePythonServer0.1")
46 ...     self.send_header("Content-Type", "text/html")
47 ...     self.end_headers()
48 ... # and finally the HTML is sent
49 ...     self.reply("<html><head><title>The Time</title></head>")
50 ...     self.reply("<body><h1>The Time</h1>")
51 ...     now = dt.datetime.now()
52 ...     self.reply("is {0:%H}:{0:%M} now".format(now))
53 ...
54 ...     self.reply("<br><br>You requested " + self.path)
55 ...     self.reply("<br></body></html>")
56 ...
57 ... # the try is so that a Control-C will shut the server down properly
58 ... >>> def run(port):
59 ...     svr = HTTPServer(('', port), serverclass)
60 ...     try:
61 ...         svr.serve_forever()
62 ...     except:
63 ...         svr.server_close()
64 ...
65 ... run(2345)

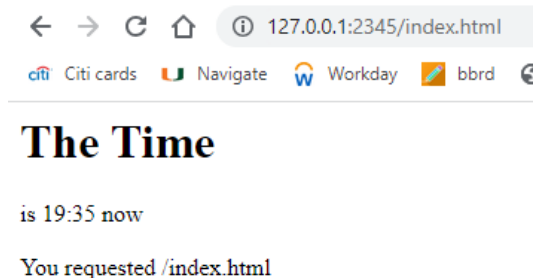
```

The `serve_forever` and `server_close` methods of `HTTPServer` do what their names suggest. `serve_forever` will carry on accepting requests and passing them to the appropriate `do_...` method until everything is stopped.

I then enter the URL `127.0.0.1:2345/index.html` in my web browser, and this (cut down a bit) is what appears in the python session:

```
connection from ('127.0.0.1', 55674)
requestline is GET /index.html HTTP/1.1
command GET
path /index.html
header Host 127.0.0.1:2345
header Connection keep-alive
... ..
header Sec-Fetch-Dest document
header Accept-Encoding gzip, deflate, br
header Accept-Language en-GB,en-US;q=0.9,en;q=0.8
```

And the web browser shows



But then immediately the Python session records a second request, this one resulting in an error message because it was for `GET /favicon.ico HTTP/1.1`. This is because a lot of web browsers, when they first contact a server, ask for a little image (favourite icon) to be shown in the tabs at the top of their window. As long as they don't get one, they will ask again with every subsequent request.

#### iv. HTML processing

There are some complexities involved in dealing with HTML documents, and there are Python modules to help out. The most basic are to do with special characters. HTML documents do not contain Unicode characters, they are limited to ASCII, but you can represent Unicode characters in HTML and have them displayed correctly in web browsers. Example, a capital Greek Sigma is supposed to look like this:  $\Sigma$ , it is not an ASCII character. This sequence: `&Sigma;` represents it in HTML, the ampersand and semicolon are both part of it. Sigma's Unicode position (code point) is 931, so `&#931;` would do the same thing. There are also characters that mean something special in HTML and have to be represented differently if they are to appear in a document, such as `<` becoming `&lt;` and `&` becoming `&amp;`. The `html5` dictionary does the reverse mapping, but for some reason you must not include the initial `&`.

```
1 >>> import html.entities as hen
2 >>> hen.entitydefs["Sigma"]
3     '\Sigma'
4 >>> hen.name2codepoint["Sigma"]
5
```

```

6 >>> 931
7     hen.codepoint2name[931]
8 >>> 'Sigma'
9     hen.html5["lt;"]
10 >>> '<'
11     hen.html5["Sigma;"]
12     'Σ'

```

Those four things are all ordinary dictionaries, so you can use `.keys()` and so on to find out what's there. It would be nice if there were an inverse for `entitydefs`, so that we could look up any Unicode character we're interested in and get its HTML version. Of course, most Unicode characters don't have an HTML representation beyond the numeric forms like `&#x6C34;` and it is not at all hard to turn a dictionary inside out, switching around keys and values.

Whole strings can be processed in one go. `html.escape` is an ordinary function that takes and returns a string. All characters that could cause misunderstanding in HTML when you want them to appear in text are converted to their safe form, e.g. `<` turns into `&lt;`. If you set the optional second parameter to `False`, quotes will be left alone. Non-ASCII characters are generally left alone. A browser might not display them as intended, but they do not cause any misunderstanding of the structure of the document.

`html.unescape` does a bit more than the opposite of that. All special character forms like `&lt;`, `&sigma;`, and `&#931;` are converted into the characters they represent.

```

1 >>> html.escape("123<345>> &6 \"or then' Σ!")
2     '123&lt;345&gt;&gt; &amp;6 &quot;or then&#x27; Σ!'
3 >>> html.escape("123<345>> &6 \"or then' Σ!", False)
4     '123&lt;345&gt;&gt; &amp;6 "or then\' Σ!'
5 >>> html.unescape("123&lt;456 &amp;&amp; &sigma;&gt;&#931;")
6     '123<456 && σ>Σ'

```

The `html.parser.HTMLParser` class reveals the structure of any string of HTML. Once you have created an instance, you feed it HTML strings. It reads whatever you give it, and calls special methods for everything it encounters. In order to do anything useful, you need to override those methods with versions that do whatever processing you need.

The constructor has one optional keyword parameter `convert_charrefs` (default `True`). If `true` then whenever a character entity such as `&lt;` or a character numeric reference such as `&#931;` is seen, it is automatically converted to the appropriate Unicode character and delivered as any other character would be. If `false` then special handler methods will be called for them instead.

The methods are:

```
handle_decl(self, s)
```

Called when a declaration is encountered. For example, when `<!doctype = html>` is seen there would be a call to `handle_decl("doctype html")`.

`handle_comment(self, s)`

Called when a comment is encountered. For example, when `<!--I wrote this on Monday afternoon-->` is seen there would be a call to `handle_comment("I wrote this on Monday afternoon")`.

`handle_starttag(self, name, atts)`

Called whenever a beginning tag like `<tr>` or `<div id="inset" kind=six>` is encountered. The `name` parameter will be the name of the tag, "div". The `atts` parameter will be a list of two-tuples containing the names and values of all of the attributes: `[("id", "inset"), ("kind", "six")]`. Note that the unnecessary quotes have been removed from `inset`.

`handle_endtag(self, name)`

Called whenever an ending tag like `<tr>` or `</div>` is encountered. The `name` parameter will be the name of the tag, "div".

`handle_startendtag(self, name, atts)`

Called whenever a tag ends with `/>`, which acts as an automatic ending, is encountered. The parameters are exactly the same as for `handle_starttag`.

`handle_data(self, s)`

Called for all the ordinary words and characters and symbols and things that appear between tags.

`handle_entityref(self, name)`

Only called if `convert_charrefs` was `False`. Called when a character reference like `&lt;` or `&Sigma;` appears. For those examples, `name` would be "lt" or "Sigma". Note that the `&` and `;` are both removed.

`handle_charref(self, form)`

Only called if `convert_charrefs` was `False`. Called when a numeric character reference like `&931;` or `&x3A3;` appears. For those examples, `name` would be "931" or "x3A3".

`close()`

Normally you wouldn't override this method, but if you need some extra processing to be performed when the end of the document is reached, put it in here. It is important to remember to call `HTMLParser.close()` afterwards.

After creating an `HTMLParser` object, you feed fragments of HTML into it using the `feed(s)` method. `s` must be a string. It doesn't matter if the string ends awkwardly, perhaps in the middle of a tag. It will save the unfinished item until more text is fed in. You don't have to feed it small fragments, it can take a whole file at once. Call the `.close()` method when everything is done.

```

1 >>> class hparse(HTMLParser):
2 ...
3 ...     def handle_starttag(self, name, atts):
4 ...         print("start", name)
5 ...         print("  ", atts)
6 ...
7 ...     def handle_endtag(self, name):
8 ...         print("end", name)
9 ...
10 ...
11 ...     def handle_startendtag(self, name, atts):
12 ...         print("start and end", name)
13 ...         print("  ", atts)
14 ...
15 ...     def handle_data(self, s):
16 ...         print("data")
17 ...         print("  ", s)
18 ...
19 ...     def handle_entityref(self, s):
20 ...         print("character entity", s, hen.entitydefs[s])
21 ...
22 ...
23 ...     def handle_charref(self, s):
24 ...         if s[0] == "x" or s[0] == "X":
25 ...             ch = chr(int(s[1 : ], 16))
26 ...         else:
27 ...             ch = chr(int(s, 10))
28 >>>         print("character reference", s, ch)
29 >>>
30
31     p = hparse()
32     p.feed("<html><div id=\"small\" type=six>hello cat</div>")
33     start html
34     []
35     start div
36     [('id', 'small'), ('type', 'six')]
37 >>> data
38 ...     hello cat
39     end div
40     p.feed("""also <img src=abc.jpg /> and &amp; &lt; &#931;
41         &Sigma; </html>""")
42     data
43     also
44     start and end img
45     [('src', 'abc.jpg')]
46 >>> data
47     and & < Σ
48         Σ
49     end html
50     p.close()

```

But

```

1 >>> p = hparse(convert_charrefs = False)
2 >>>

```



```

3 ... p.feed("""also <img src=abc.jpg /> and &amp; &lt; &#931;
4         &Sigma; </html>""")
5     data
6         also
7     start and end img
8         [('src', 'abc.jpg')]
9     data
10        and
11    character entity amp &
12    data
13
14    character entity lt <
15    data
16
17    character reference 931 Σ
18    data
19
20    character entity Sigma Σ
21    data
22
23    end html

```

Notice that you are given the spaces between special symbols as data because in HTML it is important to know whether things were separated or not. But also notice that the multi-line string, in both examples, resulted in all the spaces and newlines being included, even though the amount of white space is meaningless in HTML, only its presence or absence matters.

## v. Recognising URLs

URLs can have quite a few (six) of components:

First the *scheme*, such as HTTP, followed by `://`

Second the *net location*, such as `www.abc.com` or `xyz.org:88` followed by `/`

Third the *path* such as `library/four/index.html` followed by `;`

Fourth the *parameters*, just about anything, followed by `?`

Fifth a *query*, like `uvw=12;xyz=cat+food`, followed by `#`

Sixth a *fragment*, also just about anything.

`urllib.parse`'s `urlparser` function will split them up as a named tuple for you. For each of the six components above, anything that is absent results in an empty string. Net location is also split into hostname and port, where absence results in `None`.

```

1 >>> from urllib.parse import urlparse
2 >>> x = urlparse("http://abc.com:456/idx.html?abc=2;xyz=cat+food")
3 >>> x.scheme
4     'http'
5 >>> x.netloc
6     'abc.com:45'
7 >>> x.hostname
8     'abc.com'

```

```

9 >>> 'abc.com'
10     x.port
11 >>> 456
12     x.path
13 >>> '/idx.html'
14     x.params
15 >>> ''
16     x.query
17 >>> 'uvw=2;xyz=cat+food'
18     x.fragment
19     ''

```

Unfortunately this is of very limited use because it doesn't understand URLs in quite the way web browsers and people do. That means that you can't give it a URL as entered by a user or found in text. If you enter `abc.com/index.html` into a web browser, we all know what will happen: host name (or net location) is `abc.com` and file (or path) is `index.html`. But ...

```

1 >>> urlparse("abc.com/index.html")
2     ParseResult(scheme='',
3                 netloc='',
4                 path='abc.com/index.html',
5                 ...

```

`urllib.parse.parse_qs` will split a query into its component name, value parts. The result is presented as a dictionary where the keys are the names and the values are the values. Values are always presented as a list of strings, which strongly suggests that multiple values for one name are possible, but there is no character which when used as a separator produces a list of length more than one.

`parse_qs` understands the % encoding used to allow troublesome characters, such as #, to safely appear. `urlparser` does not. The % sign is followed by the character's two digit hexadecimal ASCII code, so space becomes %20 and # becomes %23.

```

1 >>> parse_qs("uvw=2,13&xyz=cat+food")
2     {'uvw': ['2,13'], 'xyz': ['cat food']}
3 >>> parse_qs("uvw=2,13&xyz=cat%20fo%23od")
4     {'uvw': ['2,13'], 'xyz': ['cat fo#od']}

```

When you are creating a query as part of a URL, `urllib.parse.quote` does that encoding for you. All characters apart from letters, digits, ., ~, -, and \_ are converted to their % form. `quote_plus` converts spaces into + rather than %20. `unquote` and `unquote_plus` do the opposite.

```

1 >>> quote("abc=12;xyz=cat food#175*x")
2     'abc%3D12%3Bxyz%3Dcat%20food%23175%2Ax'
3 >>> quote("abc=12;xyz=cat food#175*x")
4     'abc%3D12%3Bxyz%3Dcat+food%23175%2Ax'
5 >>>

```

```
unquote("abc%3D12%3Bxyz%3Dcat%20food%23175%2Ax")
'abc=12;xyz=cat food#175*x'
```

## 35. Network clients and servers

For the most general-purpose package, `import socket`. That gives you the same sort of interface as you get when programming in C under Unix. Some kinds of servers and clients have already been implemented as Python modules. If you want web access, `import http` instead, and you'll find that just about everything has been done for you.

If you are already familiar with network programming, Python provides a lot of minor constants and methods that I won't describe here. They allow you to do all the operations that you already know about.

To communicate over the internet, you need at least two pieces of information to describe the thing you want to communicate with. The first is a host name or IP address, and the second is a port number. All internet accessible computers have an IP address, it is usually four smallish numbers separated by dots (IPv6 is different, but still isn't seen so very much) something like "129.171.33.6". The host name is something more user friendly, like "rabbit.eng.miami.edu". There is a sort-of one-to-one mapping between host names and IP addresses, and Python automates the conversion. An IP address identifies a particular computer, but each computer could be running a large number of different services. The port number identifies which one you want, they range from very small numbers to just above 65000.

Some network services require even more. The biggest example is the web server. To tell your browser what you want to see, you provide a URL which consists of six items: protocol, host name, port number, path, resource, and application data. protocol is optional and defaults to HTTP. port number is optional and defaults to 80, path is optional and defaults to "/", it is really just a path as is always used to explicitly address a particular file, like "directory/subdirectory", resource is also optional, defaulting to "index.html", and application data is also optional. Path and resource are almost always considered together as a single thing. Protocol is separated from hostname by "://", host name is separated from port number by ":", port number is separated from path by "/", path is separated from resource by "/" again, and resource is usually separated from application data by "?". URLs can be as simple as

```
amazon.com
```

or as complex as

```
https://abc.def.org:1293/products/ink/pricing.cgi?query=indian+ink
```

If you are trying network programming out on your own, you do not need two computers, or even a real IP address. Just start up two Python sessions, and they will be able to communicate with each other exactly as though they were thousands of miles apart. When you start a server, you can either specify as

desired number, or be given one automatically. If two sessions are on the same computer, just use “127.0.0.1” as the host name or IP address.

## i. Sockets

A socket is the basic object for network connections. Just like a file object (or more accurately, an `_io.TextIOWrapper` object) it represents the ability to create a connection, or to represent the state of a connection if it is already open.

To create a socket, you must decide on the kind of connection you want. This is called the address family. The most common ones are `AF_INET` for most internet connections (it uses IPv4, where IP addresses are four numbers with dots between them), or `AF_INET6` (for the still not very common IPv6, where addresses are eight groups of four hexadecimal digits with colon between them). If you know more about networking, there are others that may be of interest, like `AF_BLUETOOTH`, `AF_UNIX`, and `AF_PACKET`. I'll assume `AF_INET` here.

Assuming we're working with `AF_INET`, you next need to decide on how communications should proceed. This is called the protocol, but does not mean quite the same as it does in a URL. There are five protocols to choose from, but we'll only look at the two most useful of them, `SOCK_STREAM` (for TCP) and `SOCK_DGRAM` (for UDP). `SOCK_STREAM` is for real connections that work as streams of bytes flowing in both directions, it refers to the TCP protocol. There are also a lot of differences between Windows and Unix based systems. In most cases I will only cover things that seem to work everywhere.

`socket` is a class from the `socket` module. As expected we create a socket object by using its constructor. The two important parameters are the address family, `AF_...` and the protocol, `SOCK_...`, e.g. `sock = s.socket(s.AF_INET, s.SOCK_STREAM)`.

To communicate using TCP, you first establish a connection between server and client. That takes a bit of work, but it is all done automatically for you. Once the connection is made you can send messages back and forth, knowing that any errors will be detected, if a transmission fails to get through, it will automatically try again for a while, but if the failure is persistent you will know about it. When communications are over, you must deliberately close the connection. One pair of sockets connects a client with a server (usually) for a single session, and can't do anything else.

Communication with UDP is much more basic. Once you have opened your socket you can use it to communicate with anyone else on the internet. Each message you send has its own specific destination attached to it, so one socket is all you really need. UDP is not designed to be reliable. It is a bit like sending a telegram or a message in a bottle. You have no way of knowing whether a message was received or not, unless the recipient successfully sends something back. This is not a bad thing. It makes high-bandwidth communications like video streaming practical. All the checking and resending done by TCP would make everything too slow.

After creating a socket, things are different depending on whether you are programming the client or the server. With TCP that is, with UDP there is no major difference between clients and servers. The usual pattern is that a server will just sit and wait for clients to connect to it. The client will send a series of requests, and the server will send back its responses. The server's behaviour is called passive. The client is the opposite, its behaviour is active. It initiates connections with a server when it wants to. It requests what it wants, and most frequently it closes the connection when it has had enough.

## ii. TCP clients

Once a socket is created, a client uses the `connect` method to open communications with a server. `connect` needs two things, but strangely insists on them both being provided in a single tuple. The first is the host name for the server, that will be a well known string like `"www.google.com"` or an IP address like `"137.212.33.91"`. The second is a port number, just an int. These are the first three steps:

```
1 >>> import socket as s
2 >>> conn = s.socket(s.AF_INET, s.SOCK_STREAM)
3 >>> conn.connect(("rabbit.eng.miami.edu", 80))
```

`s.socket` will only fail under extreme and unusual circumstances, but there are a whole lot of possible reasons for `connect` failing, each will produce a usually helpful exception. If `connect` succeeds, you are connected to the server.

Under Unix, you get two more options. The second parameter to `socket` may be bitwise orred (the `|` operator) with one or both of `socket.SOCK_NONBLOCK` and `socket.SOCK_CLOEXEC`. `NONBLOCK` means that if you try to do something that can't be done yet (that is most usually reading data that hasn't been sent yet), instead of waiting, it will immediately return. `CLOEXEC` is unlikely to be very useful in Python programming, it just means that the socket will be closed automatically if one of the `exec` system calls is used.

To transmit, you use the socket's `send` method. It takes a `bytes` object as the message to be sent, and returns the number of bytes that were successfully sent. To receive, you use the socket's `recv` method. The parameter is the maximum number of bytes you are willing to take, and the returned result is a `bytes` object. When you have finished communicating, both server and client should call the socket's `close` method.

Here is a short example of a client interacting with a web server. I have cut down the output to make it readable.

```
1 >>> import socket as s
2
3 >>> def ask():
4 ...     conn = s.socket(s.AF_INET, s.SOCK_STREAM)
5 ...
```

```

6 ...     conn.connect(("rabbit.eng.miami.edu", 80))
7 ...     conn.send(b"GET /findme.html HTTP/1.0\r\n")
8 ...     conn.send(b"\r\n")
9 ...     while True:
10 ...         bs = conn.recv(1024)
11 ...         print("-----")
12 ...         print(bs)
13 ...         if len(bs) == 0:
14 ...             break
15 ...     conn.close()
16 >>>
17
18     ask()
19     -----
20     b'HTTP/1.1 200 OK\r\nDate: Fri, 14 Jul 2023 22:17:11
21     Connection: close\r\nContent-Type: text/html\r\n\r\n
22     <html><head><title>How to find Me</title></head>\n<body>
23     ...
24     A\n<tr><td><'
25     -----
26     b'td><td>Tuesday<td>\n<tr><td><td><td><td> 2:00 to 3:15
27     ...
28     colspan=5>&nbsp;\n</table>\n</body></html>\n\n'
-----
b''

```

A few things to observe. First, it is using HTTP version 1.0 which is completely outdated. The reason for using it is that it has minimal requirements. Just a GET command. Second, the HTTP protocol demands that lines end with the two character sequence `\r\n` (ASCII 13, 10). The second `send` is also a requirement. Until you send a blank line, the server will think you are just sending more details of your request. Fourth, I didn't ask the server to keep this connection alive, so it closed the socket after its reply. Fifth, when a socket is closed, a `recv` will give you an empty `bytes` object rather than raising an exception.

To cut through the ugliness of the `bytes` objects, I'll replace the printing of the dashes and `print(bs)` with `print(bs.decode(), end = "")`. Now we can see what the response really looked like:

```

1 >>> ask()
2     HTTP/1.1 200 OK
3     Date: Sat, 15 Jul 2023 00:13:04 GMT
4     Server: Apache/2.4.56 (FreeBSD)
5     ...
6     Content-Type: text/html
7
8     <html><head><title>How to find Me</title></head>
9     <body><h2>How to find Me, Autumn 2023</h2>
10    <table border=0>
11    ...
12    </table>
13    </body></html>

```

That allows one final observation. The HTTP protocol is a bit noisy. Web servers do not just provide the information you request, they precede it with some headers that provide metadata, extra information about it. The first line is the most important and always begins with `HTTP/` followed by the version, a code number and a bit of text giving a hint at what the code means. 200 represents success, most other numbers indicate a failure of some kind. After that you get bits of detail that might be of help. The last one, `Content-Type` lets you know how the response should be interpreted. Then there is always a blank line, after which the real data follows.

To save one line of code, `socket` has a slightly convenient class method equivalent to `socket()` followed by `connect()`. It is `create_connection((host, port))`.

### iii. TCP servers

The setup for a server is only slightly more complex. Usually you will have chosen the port number you want for your service in advance, but you can just let the system find you any free port. The next step after creating the socket is to use the `bind` method to attach your socket to that port. `bind` has one parameter which is a tuple `(host, port)`, `host` should almost certainly be an empty string because it's pretty obvious that the service is going to run on this exact computer. `port` is the desired port number or zero if you don't care. After using `bind` with `port = 0`, you can use the parameterless `getsockname` method. It returns a tuple `(host, port)`, with `port` being the port number you were assigned.

The next step is to turn the port on and get it ready to accept requests. Use the `listen` method for this. Its parameter should be a small but definitely not zero int. 3 is traditionally used. It is a queue size for as-yet unserved incoming requests.

Next, assuming that your server is to be able to respond to more than one request, you would have a loop. Each time round the loop, use the `accept` method to accept a connection. It just waits until a client tries to connect, and accepts that connection. `accept` returns a tuple as its result. `[1]` identifies the client that has made the connection, `[0]` is a new socket. That is the socket that you `recv` and `send` with to communicate with the client. The original socket is left alone so that it can continue to accept more client connections.

Any serious server is going to want to be able to handle multiple requests at the same time, so a loop on its own isn't very good. If one request requires a long time to be answered, other clients may time-out waiting for the server to get back to the `accept`. To handle this, servers traditionally use subprocesses to deal with each request. But for most uses, a thread will do perfectly well, and be more efficient.

When the client session is over, close the new socket. Only close the original socket when you want to shut down the server.

This example implements a very basic server. Once it has started, it accepts connections and waits for them to send a string. It doesn't care what the string is. It then sends a reply that includes the time, and closes the connection. Remember that a string's `encode` method gives the bytes version of it. For the sample run, two clients connected, both asked "time please".

```
1 >>> import socket as s
2 >>> import datetime as dt
3
4 >>> def serve():
5 ...     conn = s.socket(s.AF_INET, s.SOCK_STREAM)
6 ...     conn.bind("", 0)
7 ...     print("I am on port", conn.getsockname()[1])
8 ...     conn.listen(3)
9 ...     while True:
10 ...         (clisock, cliaddr) = conn.accept()
11 ...         print("new connection from", cliaddr)
12 ...         bs = clisock.recv(1024)
13 ...         print("they said", bs)
14 ...         now = dt.datetime.now()
15 ...         reply = "Hello, it is {0:%H}:{0:%M}\r\n".format(now)
16 ...         clisock.send(reply.encode())
17 ...         clisock.close()
18 >>>
19
20
21     serve()
22     I am on port 12225
23     new connection from ('129.171.33.6', 30513)
24     they said b'time please\r\n'
25     new connection from ('129.171.33.6', 27487)
26     they said b'time please\r\n'
```

Beware that under Windows, Control-C does not interrupt an `accept`. This is what the clients saw:

```
1 >>> cl.ask(12225)
2     b'Hello, it is 19:28\r\n'
3 >>> cl.ask(12225)
4     b'Hello, it is 19:31\r\n'
```

To save two lines of code, there is a very slightly convenient method of the `socket` class: `create_server((host, port))` it is equivalent to `socket()` followed by `bind()` followed by `listen()`.

#### iv. UDP agents

UDP agents (for want of a better word) are simpler because they don't make connections and only need a single socket for everything.



Essentially you just create a `SOCK_DGRAM` socket and use its `bind` method to attach it to a port number. Then start using its `sendto` method to transmit things and its `recvfrom` method to receive things.

`bind` is the same as it is for TCP, give it a tuple of "" and a port number. Set the port number to 0 if you don't care what number you get. After binding, the socket's `getsockname` method will deliver the familiar tuple whose [1] is the port number.

I need two Python sessions so that there is something to talk and something to listen. `recvfrom` takes a maximum number of bytes as its parameter and returns a (message, sender) tuple. `sendto` takes a bytes object for the message to be sent, and an (IP address, portnumber) tuple for the destination. It returns the number of bytes successfully sent.

Session one:

```
1 >>> import socket
2 >>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3 >>> s.bind(("", 0))
4 >>> s.getsockname()[1]
5     53201      # that's my port number
6 >>> (msg, sender) = s.recvfrom(1024)
```

and it sits waiting there. Session two:

```
1 >>> import socket
2 >>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3 >>> s.sendto(b"Hello!\n", ("127.0.0.1", 53201))
```

Notice that because I'm not receiving, I don't even need to use `bind`. If `bind` isn't used, the first `sendto` will do an automatic bind with a port number of zero. `getsockname` will not work until a bind has been done.

Back to session one, which is no-longer waiting:

```
1 >>> msg
2     b'Hello!\n'
3 >>> sender
4     ('127.0.0.1', 65162)
```

## 36. Polling - asynchronous communication

There are many situations in which you want to be able to receive input, but do not want to have to wait for it. A common example is a program that is involved in a long computation but needs to be able to receive commands from the user from time to time, or perhaps a network service in which a real person is monitoring things, so that they can tell the system to cut off communications with an abusive user.

In many cases, multi-threading provides a solution. This section is about other options.

This is one of those areas where Windows gives a lot of trouble, so we'll start off doing things the Unix way.

## i. Unix

The `sys` module provides access to three already open files `sys.stdin`, `sys.stdout`, and `sys.stderr`. They are exactly the `stdin`, `stdout`, and `stderr` files that all Unix users are familiar with. `stdin` is by default your keyboard, `stdout` and `stderr` are by default your screen, but all three can be redirected. The Python `input` function always reads from whatever happens to be `stdin` at the moment, and `print` always writes to `stdout` unless told otherwise.

The `select` module gives us a function called `select`. Its job is to tell us whether input or output is possible without having to wait for it. It has four parameters, the first three of them I'll call `gr`, `gw`, and `gx`, the `g` stands for given. They are iterables of file-like objects. File-like objects include `sys.stdin` and its friends, actual open files, and network sockets. If you know something's Unix `fd` (the `int` that uniquely identifies any file-like object) you can just use that number instead. `stdin`, `stdout`, and `stderr` are always 0, 1, and 2 respectively. The first parameter is where you ask about the ability to read, the second is for writability, and the third for finding out if something has gone wrong.

If you have a socket called `sock`, and you want to find out whether there is data ready to be received either from that socket or the keyboard, then the first parameter would be `[sock, sys.stdin]` or `[sock, 0]`, the other two would be `[]`. `select` returns a three-tuple of lists (`rr`, `rw`, `rx`), the initial `r` stands for result. `rr` will contain all the things from `gr` that you can now take input from without having to wait. Similarly `rw` contains the writable things from `gw`, and `rx` contains the things from `gx` that have failed in some way. Most uses of `select` only involve `gr` and `rr`, input has to wait until something has sent something, but in most cases output is always possible.

The fourth parameter to `select` is a timeout. If you don't provide it, `select` becomes a blocking function, it will wait until something does become possible. Even that is useful, because you would be waiting for a number of things, any one of which would end the wait. If the timeout is 0, `select` will return immediately with just the things that were already possible. Any other value for the timeout is the maximum number of seconds to wait for something to happen. If nothing happens before the timeout expires then all three of the returned lists will be empty.

This is one of the things that just don't work under Windows. `select` will happily take a number of sockets, but will not accept `sys.stdin`.

Here is a rather dull use of that.

```
1 >>> import sys
2 >>> import select
3
4 >>> def kbwait():
5 ...     while True:
6 ...         (rr, rw, rx) = select.select([sys.stdin], [], [], 5)
7 ...         if sys.stdin in rr:
8 ...             s = input()
9 ...             print("you said", s)
10 ...         else:
11 ...             print(".")
```

Running that, you would see a very slow progression of dots, one every five seconds. When you press a keyboard key, nothing seems to happen. All systems do buffering on keyboard input that makes it wait until you have pressed enter. Without that, the backspace key would be useless. As soon as you do press enter, the whole line you typed is reported and the function carries on waiting for more.

Here is a bigger example, a UDP service that will take a query of the form "A B" from anyone, A and B are space separated decimal numbers. The service will respond with whatever the value of  $A \times B$  is. It will continue until you type stop. We'll assume that `sys`, `select`, and `socket` have already been imported.

```
1 >>> def run():
2 ...     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3 ...     s.bind(("", 0))
4 ...     print("Tell our users to use port", s.getsockname()[1])
5 ...     while True:
6 ...         (r, w, x) = select.select([sys.stdin, s], [], [], 0.1)
7 ...         if sys.stdin in r:
8 ...             cmd = input()
9 ...             if cmd == "stop":
10 ...                 break
11 ...             elif s in r:
12 ...                 (msg, sender) = s.recvfrom(1024)
13 ...                 qparts = msg.decode().split()
14 ...                 reply = str(int(qparts[0]) * int(qparts[1]))
15 ...                 s.sendto(bytes(reply, "utf8"), sender)
```

When this is running, you won't see any output until you type stop. Annoyingly, when you type stop, it will appear twice. I don't really know what causes that.

Keep in mind that you are not limited to waiting for just one or two things. Those lists of file-like objects can be quite long. Unix itself imposes a limit.

To ask this service a question, these four lines will do the trick. 45266 is of course the port number that the service reported when it started.

```
1 >>> import socket
```

```

2 >>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
3 >>> s.sendto(b"12 240\n", ("127.0.0.1", 45266))
4 >>> s.recvfrom(1024) [0]
5     b'2880'

```

If you don't even want to wait for the user to press enter, but just take whatever is pressed as soon as it is pressed, there doesn't seem to be any official Python module, but based on some web searches, I managed to make one. If you are using rabbit, you just have to import `specialkbd`. If you are not using rabbit, I have pasted in the entire file just below the next bit.

`specialkbd` defines a `kbd` class. You should not use its constructor, but call the module's ordinary `get_kbd()` function instead. `get_kbd()` does return an instance of the `kbd` class, but ensures that no more than one instance is created, which would be bad.

The `kbd` object has four methods. Before trying to catch any keyboard characters without waiting for an enter, you must call `special_mode()`, and when you have finished, you must call `normal_mode()`. If you forget `normal_mode` your terminal will not behave properly. Of course, you can always use the Idle prompt and call `normal_mode` by hand. Exiting from Idle *should* restore everything to normal too.

`kbd` has `__enter__` and `__exit__` methods, so you can use it in a `with` statement and forget about `normal_mode`. See the example coming soon.

Once your terminal is in special mode, you have two methods to choose from: `get_key()` and `get_key_no_wait()`. `get_key()` waits until any key is pressed, it doesn't matter whether it is enter or not. Normally it will just return that character as a string of length one. But some keys, such as the arrow keys, produce a small sequence of characters which you need to see all at the same time, in that case it returns a list of characters.

Putting your terminal in special mode also turns off the automatic echo mechanism. Normally, when you press a keyboard key, the character appears on your screen right away. With echo off that doesn't happen. You don't see anything unless you print it. If you don't want this behaviour, `special_mode` has an optional `doecho` parameter. If you set it to `True`, echoing will happen as normal.

If you do a normal `input()` while in special mode, it will still work as usual except that echoing will be turned off (unless you set `doecho` to `True`).

Note from the example below that it even captures Control-C, so you need to be careful. Note also that I am using `repr()` inside `print` to make everything visible. without `repr`, enter and Control-C would be invisible.

```

1 >>> import specialkbd as spk
2
3 >>> def run():
4 ...     with spk.get_kbd() as k:
5 ...

```

```

6 ...     k.special_mode()
7 ...     while True:
8 ...         c = k.get_key()
9 ...         print(repr(c))
10 ...        if c == "x":
11 >>>            break
12
13     run()
14     'W'           # I pressed W
15     '7'           # I pressed 7
16     '\r'         # I pressed enter. '\r' was returned, not '\n'
17     ['\x1b', '[' , 'A'] # I pressed the up-arrow
18     '\x03'       # I pressed Control-C, that is ASCII 3
19     'x'           # I pressed x

```

`get_key_no_wait()` is exactly the same as `get_key()`, except for one thing, it doesn't wait for anything at all. If a key hasn't been pressed already, it immediately returns `None`.

This is not compatible with the `select` method. If you want to wait for input from either of two sockets or the keyboard, without needing enter to be pressed, you would need to combine them in a loop. First try `get_key_no_wait`, if that doesn't give you anything try `select` with a zero timeout, if that doesn't give you anything, take a little sleep and go round the loop again. The little sleep is important, without it, you will just consume all the CPU's computational power getting nowhere.

```

1 >>> while True:
2 ...     ch = k.get_key_no_wait()
3 ...     if ch != None:
4 ...         ... # got something from the keyboard, deal with it.
5 ...     else:
6 ...         (rr, rw, rx) = select.select([sock1, sock2], [], [], 0)
7 ...         if rr != []:
8 ...             ... # a socket is ready, deal with it.
9 ...         else:
10 ...             time.sleep(0.05)

```

This is the code for `specialkbd.py`. I expect it to work on any version of Unix, but have only had a chance to test it on FreeBSD.

```

import termios
import fcntl
import sys
import os
import queue
import sys

class kbd:

    the_one = None

    def __init__(self):
        if kbd.the_one != None:

```

```

    raise Exception("you must not create two kbd objects")
self.stin = sys.stdin.fileno()
self.saved_flags = fcntl.fcntl(self.stin, fcntl.F_GETFL)
self.saved_tattrs = termios.tcgetattr(self.stin)
self.echobits = termios.ECHO | termios.ECHONL
self.backed_up = queue.Queue()
self.started = False
kbd.the_one = self

@classmethod
def get_kbd(thisclass):
    if kbd.the_one == None:
        kbd()
    return kbd.the_one

def __enter__(self):
    return self

def __exit__(self, etype, eval, etrace):
    self.normal_mode()
    return etype == None

def special_mode(self, doecho = False):
    tattrs = list(self.saved_tattrs)
    tattrs[0] &= ~ (termios.IGNBRK | termios.BRKINT | termios.PARMRK
                  | termios.ISTRIP | termios.INLCR | termios.IGNCR
                  | termios.ICRNL | termios.IXON )
    tattrs[1] |= termios.OPOST | termios.ONLCR | termios.ONLRET
    tattrs[2] &= ~ (termios.CSIZE | termios.PARENB)
    tattrs[2] |= termios.CS8
    tattrs[3] &= ~ (termios.ICANON | termios.ISIG | termios.IEXTEN)
    if doecho:
        tattrs[3] |= self.echobits
    else:
        tattrs[3] &= ~ self.echobits
    termios.tcsetattr(self.stin, termios.TCSANOW, tattrs)
    self.started = True

def normal_mode(self):
    if not self.started:
        return
    termios.tcsetattr(self.stin, termios.TCSAFLUSH, self.saved_tattrs)
    fcntl.fcntl(self.stin, fcntl.F_SETFL, self.saved_flags)
    self.started = False

def correct(self, c):
    if c == "\x03":
        raise KeyboardInterrupt
    if c == "\r":
        return "\n"
    return c

def deal_with(self, cs):
    if len(cs) == 0:
        return None
    if len(cs) == 1:
        if cs[0] == "":
            return None
        return self.correct(cs[0])
    if cs[0] == "\x1B":
        return cs
    for c in cs[1:]:

```

```

        self.backed_up.put(self.correct(c))
    return self.correct(cs[0])

def get_key_gen(self, block):
    if not self.backed_up.empty():
        return self.backed_up.get()
    if block:
        fcntl.fcntl(self.stin, fcntl.F_SETFL, self.saved_flags & ~ os.O_NONBLOCK)
    else:
        fcntl.fcntl(self.stin, fcntl.F_SETFL, self.saved_flags | os.O_NONBLOCK)
    keys = []
    try:
        keys.append(sys.stdin.read(1))
        if block:
            fcntl.fcntl(self.stin, fcntl.F_SETFL, self.saved_flags | os.O_NONBLOCK)
            c = sys.stdin.read(1)
            while len(c) > 0:
                keys.append(c)
                c = sys.stdin.read(1)
        finally:
            fcntl.fcntl(self.stin, fcntl.F_SETFL, self.saved_flags)
    return self.deal_with(keys)

def get_key(self):
    return self.get_key_gen(True)

def get_key_no_wait(self):
    return self.get_key_gen(False)

def get_kbd():
    return kbd.get_kbd()

def stringy(s):
    r = "\""
    for c in s:
        if c.isalnum():
            r += c
        else:
            r += "[" + str(ord(c)) + "]"
    r += "\""
    return r

def getstring(echo):
    kb = get_kbd()
    s = ""
    while True:
        c = kb.get_key()
        if c == "\n":
            return s
        if c == "\x08":
            if s != "":
                if echo:
                    print(" \x08", end = "")
                    sys.stdout.flush()
                s = s[0 : -1]
        elif type(c) == str:
            s += c

def run(echo = True):
    k = get_kbd()
    k.special_mode(echo)
    try:

```

```

while True:
    s = getstring(echo)
    print("it was:", stringy(s))
    if s == "stop":
        break
except KeyboardInterrupt:
    print()
    pass
except:
    raise
finally:
    k.normal_mode()

```

## ii. Windows, the pynput module

Under Windows, the `select` module's `select` function tries to work like the Unix version in the subsection above, but unfortunately it only works on sockets and is unable to detect keyboard input being ready.

There is a module called `msvcrt`, but I have not found any circumstances under which it works at all. Web searches reveal that it is known not to work with Idle, but it doesn't seem to work with anything else either.

There is another module called `pynput` which does provide the necessary functionality. Sadly it doesn't come with the Python distribution, so you need to install it yourself. See the earlier section on modules and packages, it has a subsection for installing extras.

This will get us started:

```

1 >>> from pynput import keyboard as kb
2 >>> import time
3
4 >>> def down(key):
5 ...     print("press", repr(key))
6
7 >>> def up(key):
8 ...     print("release", repr(key))
9
10 >>>
11 ... def run():
12 ...     L = kb.Listener(on_press = down, on_release = up)
13 ...     L.start()
14 ...     time.sleep(15)
15 ...     L.stop()

```

The `Listener` class is a subclass of `Thread`, so nothing happens until its `start` method is called. The parameters are two functions which will be called automatically so long as the `Listener` is running whenever a keyboard key is pressed or released. This thread is able to have a `stop` method because it was programmed to keep listening for a command to stop. The `Listener`'s functions continue to be called for every keyboard event even when another application has



focus. As I'm typing into Word now, my key-presses are being reported in the Idle window. Here is a run:

```
1 >>> run()
2     release <Key.enter: <13>>
3     press 'n'
4     release 'n'
5     press '5'
6     release '5'
7     press 'p'
8     release 'p'
9     press ','
10    release ','
11    press <Key.up: <38>>
12    release <Key.up: <38>>
13    press <Key.shift: <160>>
14    press 'A'
15    release 'A'
16    release <Key.shift: <160>>
17    press <Key.ctrl_l: <162>>
18    press '\x01'
19    release '\x01'
20    press '\x02'
21    release '\x02'
22    release <Key.ctrl_l: <162>>
23    press <Key.esc: <27>>
24    release <Key.esc: <27>>
```

What I typed was exactly little n, digit 5, little p, comma, up-arrow, capital A, control-A, control-B, esc. As you can see, sometimes it also captures the enter key being released after the command to run was typed, but that doesn't happen very often. You will also notice that even though the control and shift buttons have their usual effect on the characters returned ('\x01' means the character code with ASCII code 1, which is control-A), we also get up and down events for the shift and control keys. Even control-C gets captured and converted to '\x03', but once the sleep expires an exception is raised for it.

It is not difficult to stop receiving other applications' key presses in the Idle or other Python window. You need to install another non-standard package called `pywin32`. Just follow the instructions from just over a page ago for installing `pynput` but say `pywin32` instead of `pynput`. We don't want `pywin32` itself, but the `win32` package that comes with it, and specifically the `win32gui` module within that.

There is a function called `GetForegroundWindow`, it just returns an int, but that int uniquely identifies the window that currently has focus to the Windows system. It is probably safe to assume that when your own module is being loaded, it is in the window that has focus. Record the window identification when the module is loaded, and make the up and down functions make sure it hasn't changed.

```

1 >>> mywin = wg.GetForegroundWindow()
2
3 >>> def down(key):
4 ...     if mywin == wg.GetForegroundWindow():
5 ...         print("press", repr(key))

```

The up and down methods receive a single object for every keypress. For a normal key it seems to be a string, but isn't. It is a `keyboard._win32.KeyCode` object. For a special key (enter, arrows, esc, shift, page down and so on) it is a member of an enum called `Key`. You can see all the different enum values you might get if you add

```
print(dir(key))
```

to either the up or down function. The `KeyCode` objects for normal keys look like strings even when you print them with `repr`, but if they are from the numeric keypad while num lock is on, they look like `<100>`, where 100 is a unique scan code for the key, in this case the 4 key.

This function will take any key value as passed to up or down, and if it corresponds to a proper typable key it will return the character that it represents. If the `repr` of the key is of length 3, it will be like 'X', and we just want that X from the middle. If it is longer, it will be like '\x05' (for control-e) where the 05 is the hexadecimal version of its ASCII code. Under windows, pressing enter delivers ASCII code 13, which is not '\n', so that needs converting too.

```

1 >>> def decode(key):
2 ...     if type(key) == KeyCode:
3 ...         r = repr(key)
4 ...         if len(r) == 3:
5 ...             return r[1]
6 ...         return chr(int(r[3:5], 16))
7 ...     vk = key.value.vk
8 ...     if vk == 13:
9 ...         return "\n"
10 ...     elif vk in [32, 27, 9, 8]: # space, esc, tab, backspace
11 ...         return chr(vk)
12 ...     return ""

```

If either the up or down function returns `False`, it will immediately stop the listener and return things to normal. With that, a common technique is to designate a special key to be the stop command, perhaps esc:

```

1 >>> def up(key):
2 ...     print("release", repr(key))
3 ...     if key == kb.Key.esc:
4 ...         return False

```

The way the `run` function is written in the example isn't very good. If anything goes wrong with the program, there is no way you can return the keyboard to normal operations, and how long should the sleep be? The `Listener` object

implements the `__enter__` and `__exit__` methods, so we can do much better using a `with` statement.

```
1 >>> def run():
2 ...     with kb.Listener(on_press = down, on_release = up) as L:
3 ...         L.join()
```

`join()` just waits for the thread to finish, which happens when the designated key, `esc`, is typed.

Now we can get back to the idea of a controllable network service. As keyboard events always come in press-release pairs, we'll ignore the releases, and we'll use a `queue.Queue` to deliver things from the `down` function to the server. I've put it together as a class. Just the outline first:

```
1 >>> class server:
2 ...
3 ...     def __init__(self):
4 ...         self.eventq = queue.Queue()
5 ...
6 ...     def accept_key(self, key):
7 ...         self.eventq.put(key)
8 ...
9 ...
10 ...     def run(self):
11 ...         with kb.Listener(on_press = down,
12 ...                        on_release = up) as L:
13 ...             self.serve()
14 >>>
15 service = server()
16 >>>
17 ... def down(key):
18 ...     service.accept_key(key)
19 >>>
20 ... def up(key):
21 ...     pass
22 >>>
23 service.run()
```

The part left out of the class definition is the `serve` method. It is quite large, but it uses things we have seen before, so we don't need everything here. It creates two TCP sockets to provide two different services at the same time.

```
1 ...     conn1 = s.socket(s.AF_INET, s.SOCK_STREAM)
2 ...     conn1.bind(("", 0))
3 ...     print("conn1 is on port", conn1.getsockname()[1])
4 ...     conn1.listen(3)
5 ...     # and exactly the same for conn2.
```

Then we have a string that builds up all the keyboard characters received, an infinite loop for the services, and a variable to note whether anything has happened or not.

```

5 ...     typed = ""
6 ...     while True:
7 ...         acted = False

```

Next a sequence of just a few things. `select` followed by a `for` loop to deal with any new server requests, then look to see if any characters are in the queue, that needs to be in a `try`: because an empty queue causes an exception, and finally a little sleep before going round the loop again if nothing has happened.

```

1 ...     (r, w, x) = se.select([conn1, conn2], [], [], 0)
2 ...     for conn in r:
3 ...         # handle request on conn
4 ...         acted = True
5 ...     try:
6 ...         key = self.eventq.get_nowait()
7 ...         # deal with the keypress in key
8 ...         acted = True
9 ...     except queue.Empty:
10 ...         pass
11 ...     if not acted:
12 ...         time.sleep(0.05)

```

Handling the request on `conn` is the usual thing

```

1 ...     (clisock, cliaddr) = conn.accept()
2 ...     # print which server it is and cliaddr
3 ...     bs = clisock.recv(1024)
4 ...     # compose reply
5 ...     clisock.send(reply.encode())
6 ...     clisock.close()

```

Finally, dealing with the keypress is a bit complicated. Every normal character is added to the string `typed` that we are building up. When `enter` is received, we look at that string, and if it is the command `"stop"` we stop. Otherwise `enter` makes us start again with a new string. For a normal keypress, we receive a `KeyCode` object, and they do not provide a helpful method for finding out which key was pressed. So we'll rely on the fact that their `__str__` and `__repr__` methods just return the key's symbol surrounded by single quotes.

```

1 ...         if key == kb.Key.enter:
2 ...             if typed == "stop":
3 ...                 return
4 ...             typed = ""
5 ...         elif type(key) == kb._win32.KeyCode:
6 ...             thekey = str(key)
7 ...             if thekey[0] == "'" and thekey[-1] == "'":
8 ...                 typed += thekey[1:-1]

```

And keep in mind that this server will also receive keystrokes intended for other applications.

`pynput.keyboard` can also control the keyboard, or more exactly it can fool the software that responds to the keyboard. You can simulate keys being pressed and released. You need to create a `Controller` object and use its methods. The most basic are `press` and `release`. Both take a `Key` value (`Key.up`, `Key.space`, etc) or a one character string to specify the key whose action is to be simulated. `tap(k)` is equivalent to `press(k)` immediately followed by `release(k)`.

```
1 >>> from pynput import keyboard as kb
2 >>> c.press("A")
3     A
4 >>> c.release("A")
5 >>> c.press(kb.Key.up)
6 >>> c.release(kb.Key.up)
7 >>> c.tap(kb.Key.enter)
```

Much more convenient is `type`. It just takes a string and creates all the presses and releases required to type it. The following gives me enough time to move the mouse from Idle back to my Word document and click to select it. The string `hello` followed by a newline does get typed into the document. But strangely, if I leave the mouse in the Idle window, the string does appear, but it has absolutely no effect, Idle is unaware of it.

```
1 >>> time.sleep(5); c.type("hello\n")
```

The `pynput` module can also has a `mouse` class, which can control or read the mouse in the same way as the keyboard. To control the mouse, you need to create a `Controller` object, which has an attribute called `position` (an `x, y` tuple) that can be both read and written, and methods `press`, `release`, and `click`. The methods all take a `mouse.Button` object to indicate which mouse button is wanted. `click` also takes as `int`, 1 for single click, and 2 for double click.

On this computer the recycling bin icon is at the top left of the screen. These five statements report where the mouse was to start with, then move it on top of that icon, then double click, so my recycling bin is opened.

```
1 >>> from pynput import mouse
2 >>> ctrl = mouse.Controller()
3     print("mouse was at", ctrl.position)
4 >>> mouse was at (1330, 664)
5 >>> ctrl.position = (35, 35)
6 >>> ctrl.click(mouse.Button.left, 2)
```

To listen to the mouse, other than just asking its position, you need to create a `Listener` object. The constructor has three parameters. `on_move` is the function to be called every time any mouse movement is detected, `on_click` is badly named, it is called whenever a mouse button is pressed or released, there is no separate reporting of clicks, and `on_scroll` is called when you use the scrolling wheel.

The `on_move` function has two parameters, `x` and `y`, that report the position the mouse has moved to. The `on_click` function has four: `x`, `y` for the position the click happened at, `button` for which button was clicked (`mouse.Button.left`, `right`, etc), and `pressed`: `True` if pressed, `False` if released. The `on_scroll` function has four parameters, `x` and `y` for where the mouse is, and `dx` and `dy` for the direction of scrolling. `dx` and `dy` are not well documented. On my computer `dx` is always 0 (do mice ever have horizontal scroll wheels?) and `dy` is 1 for down and -1 for up. So this bit of code tells you everything that ever happens to the mouse.

```
1 >>> def moved(x, y):
2 ...     print("moved to", x, y)
3
4 >>> def clicked(x, y, button, pressed):
5 ...     act = "press" if pressed else "released"
6 ...     print(button, act, "at", x, y)
7
8 >>> def scrolled(x, y, dx, dy):
9 ...     direc = "up" if dy > 0 else "down"
10 ...     print("scrolled", dir, "from", x, y)
11
12 >>>
13 >>> L = mouse.listener(moved, clicked, scrolled)
14     L.start()
15     moved to 1104 703
16     moved to 1104 704
17     moved to 1104 705
18     moved to 1104 705
19     Button.left press at 1104 705
20     Button.left released at 1104 705
21     scrolled up from 1104 705
22     scrolled up from 1104 705
23     moved to 1104 705
24     moved to 1103 706
     L.stop()
```

### iii. Windows, the dysfunctional keyboard module

There is another module called `keyboard`, which has very serious defects and also does not come with the Python installation. If you want to get it, you must first make sure `pip` is installed as for `pynput`, and navigate to that same folder. This time the command is:

```
Scripts\pip install keyboard
```

The `read_key` function is moderately useful, but it has incurable defects:

```
1 >>> import keyboard
2 ... while True:
3 ...     k = keyboard.read_key()
4 ...     print("It was \"", k, "\"", sep = "")
5 ...     if k == "x":
6 ...
```

```
7         break
8     It was "enter"
9     It was "n"
10    It was "n"
11    It was "5"
12    It was "5"
13    It was "p"
14    It was ", "
15    It was ", "
16    It was "up"
17    It was "up"
18    It was "shift"
19    It was "A"
20    It was "shift"
21    It was "ctrl"
22    It was "a"
23    It was "a"
24    It was "b"
25    It was "b"
      It was "ctrl"
```

The extra parts in the `print` are there to make it absolutely clear exactly what we are seeing. What I typed was exactly: little n, digit 5, little p, comma, up arrow, capital A, control-A, control-B. I didn't repeat anything.

There are some immediately apparent oddities:

1. It even picks up the pressing of `enter` from the end of the command before it was even running.
2. Nearly every keypress is duplicated but sometimes it misses (we only saw one "p", on line 12), the duplication is because it reports two events, one when the key is first pressed, and the other is when it is released. If you hold a key down for more than a second or so, you get repeated reports of that key. Not surprising, holding a key down is what we always do for repetition.
3. But it gives no indication of whether it is reporting a press or a release. Doesn't sound like a problem until you combine the effect of holding a key down with the fact that it sometimes misses an action. When we see the same key repeated, we can't really tell what happened.
4. For a capital letter we see three reports: shift being pressed, the capital letter, and shift being released. It turns out that the non-repeated A was just an instance of item 2, they usually are repeated.
5. A capital letter is reported as a capital letter, "A" rather than "a", but a control-letter is left alone, for these you have to pay attention to the "ctrl" presses and releases, some of which might be missed.
6. While you're actually typing control or capital letters quickly enough, it doesn't give repeated reports for the control or shift keys even though they are being held down. But if you hold down control or shift but pause for a little while before typing anything, you do get repeated control or shift reports, and they can happen an odd number of times, so there really is no way of telling "a" and "control-a" apart.

7. You can't see it in the example, but it even reports keyboard events from other programs. I've left it running and it is still reporting the events for me typing this in to Word.
8. Control-C does actually stop this loop, even though sometimes it seems that it has failed to. Control-N means something to Idle, so when you type control-N that gets reported in the usual way, but also a little dialogue box appears and takes keyboard focus. Control-C only stops Idle when it is in command of the keyboard.
9. It does not have a time-out. If you call `read_key` your program is going to be blocked until a key is pressed.

There is a solution for one of the problems. There is another function called `is_pressed`. It tells you whether or not a particular key is currently being pressed. So we can completely ignore and "ctrl" or "shift" events and explicitly ask about those keys for every report:

```
1 ...     k = keyboard.read_key()
2 ...     if k == "ctrl" or k == "shift":
3 ...         continue
4 ...     s = ""
5 ...     if keyboard.is_pressed("ctrl"):
6 ...         s += "ctrl "
7 ...     if keyboard.is_pressed("shift"):
8 ...         s += "shift "
9 ...     s += k
```

There is an alternative way of doing things that you might find with a search. It is more complex, but also more flexible. It uses functions with names like `on_press`, `on_release`, `hook`, and `unhook`. But it has one serious disadvantage, it just plain doesn't work. With some effort you can get something out of it, but it can't be stopped. You will not be able to use the keyboard for anything else, all you can do is exit from Idle and start again. But in case you don't mind about that, here is a brief description.

`keyboard.hook(fn)`

Every time any key is pressed or released, `fn` will be called. It will be given one parameter of type `keyboard.KeyboardEvent`, which has five useful attributes:

`event_type`: a string, "up" or "down"  
`is_keypad`: True if the key is on the numeric keypad at the right  
`name`: the string for the key, "y", "shift", "7", "up", "enter", etc.  
`scan_code`: an int that is unique to the key, regardless of shift, ctrl, etc.  
`time`: a float, exactly when the event occurred.

`hook` also has an optional keyword parameter `on_remove`. `on_remove` should be a parameterless function, and it will be called automatically when this hook is removed, or at least it is supposed to be.

`keyboard.on_press(fn)`

Exactly the same as `hook` except it is only called for "down" events.



```
keyboard.on_release(fn)
```

Exactly the same as `hook` except it is only called for "up" events, but the system can't take two hooks at once.

```
keyboard.unhook_all(fn) and
```

```
keyboard.unhook(fn)
```

Are where it all fails. They are supposed to undo everything and return your keyboard to normal use, but they don't.

If you go ahead with this, the best way to handle things is probably to make your hook function add all the events to a `queue.Queue`, perhaps after pairing presses and releases to make a single *something\_typed* event and filtering out shift and ctrl events, and the rest of the program can get its input in a relatively normal way from that queue.

The `keyboard` module does allow significant mischief to be performed. A program running in the background and using `read_key` could record every keypress, capturing passwords and everything. There is also the `send` function. It, and a few other related functions, makes the system think that a particular key has been pressed:

```
1 >>> for i in range(15):
2 ...     time.sleep(1)
3 ...     keyboard.send("x")
```

that just results in `xs` appearing in my Idle window. Until I switch back to Word. Then it starts inserting `xs` into this document.

## 37. `asyncio` - more asynchronous communication

The `asyncio` package provides another workable way of dealing with unpredictable events. It only works well when you need to set a few tasks in motion, then do nothing until they have finished, then gather in their results. This is the situation with both web servers and web spiders (sometimes called crawlers). Most search engines rely on web spiders, they explore a vast number of web sites, downloading all their pages and indexing their key words. That is the sort of thing `asyncio` is good at. Once a spider has sent out a request for a web page, it has absolutely nothing to do until the results come back. Somebody else is doing all the work, the internet is transmitting things and the web servers are handling its requests. A web spider could send out a vast number of URL requests before getting responses from any, it just needs a way of keeping track of everything and watching for the results.

That is the key thing: doing nothing while waiting for results. If you have to work to produce those results, this isn't going to help.

The key concept is that of the coroutine. A coroutine is like a normal function but it is capable of just going to sleep to wait for something to happen. All systems

provide a way for functions to sleep, but the difference here is that when one coroutine goes to sleep, another can do some work. A bit like threading really, but it all happens in one thread and puts a lot less strain on your computer's resources.

Beware: sometimes generators are called coroutines. They are similar ideas but definitely not the same thing in Python.

## i. Coroutines

The whole system employs a manager that starts coroutines as requested. The first runs until it goes to sleep, at which point the manager regains control. It keeps a note of what event the coroutine is waiting for, then starts another. Every time a coroutine goes to sleep, the manager looks at all the events that have happened, pairs them with the sleeping coroutines, and puts those that can now wake up on the list of coroutines that are waiting their turn. The "co" in coroutine is because they are cooperating. Things only work if all of them voluntarily go to sleep after doing a little work.

We are about to explore an `asyncio`-based web spider. At the beginning, it won't really touch the internet. Instead we'll have our own special function to fake everything. It will receive a URL, sleep a while to simulate network delays, then return some mostly random response.

```
1 >>> import asyncio as a
2 >>> alphabet = "abcdefghijklmnopqrstuvxyz      "
3
4 >>> async def get_web_pg(URL):
5 ...     print("requesting", URL)
6 ...     await a.sleep(random.randint(10, 50) / 13)
7 ...     doc = "".join(random.choices(alphabet, k = 140))
8 ...     return URL + "=>" + doc
9
10 >>>
11     print(a.run(get_web_pg("www.abc.com/def.html")))
12     requesting www.abc.com/def.html
13     www.abc.com/def.html=>p  bovaexlfzgzjckn ozta vl f j
```

First notice that the definition begins with `async def` instead of the usual `def`. That is what tells Python to treat it as a coroutine. Next notice that instead of the usual `time.sleep(...)`, we have `await a.sleep(...)`. `time`'s `sleep` function knows nothing about coroutines, so calling it would not cause a suspension and allow other things to happen. `asyncio`'s `sleep` is a *bit* like a coroutine itself. When started it immediately puts itself to sleep, telling the manager that the passage of ... seconds is the event that should wake it up. When it does wake up, it just exits. `await` is how a coroutine says "put me to sleep until this happens". Third, notice that we don't just call a coroutine in the normal way for functions, a call to `a.run` has slipped in there.

A technicality: an `await` statement can only appear inside a coroutine, i.e. after an `async def`. If you use one elsewhere then, at least at the moment, you get a very poorly chosen error of “incomplete input”. Also `a.sleep` has an optional second parameter `result`. `x = await a.sleep(3, 999)` will set `x` to 999 after a three second wait.

The `async def` and `a.run` are central to what is going on here. `async def` changes a function into a coroutine. When you call a coroutine in the normal way, it doesn't run at all. Instead it returns, as though it were its own return value, an object that refers to the coroutine. Technically speaking it is this object that I should be calling the coroutine, not the function-like thing at all. When this object is given to `asyncio.run`, the system creates the “event loop”. The event loop is the thing that I referred to as a manager earlier. It keeps track of all the coroutines, gives them their turns at running, and monitors the real world activities that they care about. The coroutine is given to the event loop, and it will start to run just about immediately.

The event loop controls everything, you only want to have one of them. That means that a program should only use `asyncio.run` once, and only one coroutine can be started that way. That one coroutine is called the *main* coroutine, like the function called `main` in C++, C, and Java. The main coroutine controls everything, and it creates whatever other coroutines that might be wanted by using the `asyncio.create_task` method.

The `asyncio` system deals with both *tasks* and coroutines, and the distinction can be confusing. That is partly because there isn't much of a distinction to be made. The only things that can go on the event loop and be run are tasks, but a task is just an object that contains a single coroutine and some information about it. Many functions and methods will accept both coroutines and tasks as parameters, and if you give them a coroutine they will just automatically create a task around it. `a.run` and `a.create_task` are both like that. `a.run` starts the main task, waits for it to complete, then returns its result as its own. If the main coroutine starts any other coroutines, it must wait for them to complete. When the main task exits, the event loop is over.

Back to the example. So far we are not making good use of coroutines, we only create one, and it doesn't do very much. The basis of the example was a web spider that wants to get a number of URLs at once, so that's what we'll do now.

Coroutines are cheap and light-weight, there is nothing wrong with having a lot of them. We will have a main coroutine that creates as many coroutines as there are URLs that it needs to fetch. `get_web_pg` will remain exactly as it is, but we'll use something else as the main coroutine. To keep things clear, I'll call it `main`. Its one parameter will be a list of the URLs that are wanted. Instead of returning the (simulated) contents of the web page, it will print them as they become available.

```
1 >>> async def main(URL_list):
2 ...     tasks = [a.create_task(get_web_pg(u)) for u in URL_list]
3 ...     for t in tasks:
4 ...
```

```

5 ...     data = await t
6         print(data)
7 >>>
8 >>> URLs = [ "www.abc.com", "xyz.com/data", "cde.org/x.html" ]
9         a.run(main(URLs))
10        requesting www.abc.com
11        requesting xyz.com/data
12        requesting cde.org/x.html
13        www.abc.com=>byu yn  c gxkr  h es of  bvoyy
14        xyz.com/data=>fd rwzrcdk rv prsxbswkjl v fk j
        cde.org/x.html=>fmamyfvb dvk  lolcy nnasvwwds

```

main uses a list comprehension to build a list of all the needed tasks. For every value in `URL_list`, it uses `get_web_pg` to create a coroutine to fetch it, then creates a task around it. The task is able to run as soon as `a.create_task` returns, but it won't get a chance to until `main` awaits something. The next part is a bit clumsy, but it's good enough for a start. `main` just waits for each task in turn. If the thing you `await` is a task, `await` returns the task's coroutine's return result. Once `main` does its first `await`, all the other tasks will run in quick succession, when one reaches its `await`, the next in line gets its turn. The tasks will all finish in random order, but because of the `for` loop, we won't see any of that until `tasks[0]` has finished. It would be nice to avoid all that dead time and see results as soon as they arrive.

The function `a.wait` takes a list, set, or dictionary of tasks and waits until all of them have finished. Almost. `a.wait` could take a long time, and coroutines with time consuming operations block all other coroutines from running. What `a.wait` really returns is a task-like object (called an *awaitable*), and you have to `await` it to get the result. In this form, `a.wait` would be no better for us than the `for` loop we've already got. Fortunately it takes an optional second parameter that modifies its behaviour.

The second parameter to `a.wait` is surprisingly required to be a keyword parameter, `return_when`. It can have one of three values: `a.ALL_COMPLETED`, that's the default and it means wait until every task in the list is done. `a.FIRST_COMPLETED` means stop waiting as soon as any task is done. `a.FIRST_EXCEPTION` means wait until any task has an uncaught exception. A task that has an uncaught exception is also considered to be done, so we don't have to worry about the inability to combine the options.

The *awaitable* that `a.wait` returns eventually returns a two-tuple of sets. The first is the set of tasks that are now done, and the second is the set of those that are not done yet. That means that after `a.wait` it does make sense to go through the first set with a `for` loop to print all their results, then just do another `a.wait` on the second set. Repeat until the second list is empty. Here is the new `main`:

```

1 >>> async def main(URL_list):
2 ...     tasks = [a.create_task(get_web_pg(u)) for u in URL_list]
3 ...     while tasks:
4 ...

```

```

5 ...     (done, tasks) =
6 ...         await a.wait(tasks,
7 ...             return_when = a.FIRST_COMPLETED)
8 ...     for t in done:
           print(t.result())

```

As conditionals consider empty things to be false, the `while tasks` will catch both an empty initial list and an empty set of remaining tasks. A task's `result` method returns the task's coroutine's return result if it is done, and if the task is not done it raises an `InvalidStateError` exception. Also `result` is a bit dangerous because if a task was ended because of an uncaught exception, its `result` method will re-raise that exception. Fortunately there is also an `exception` method that returns (not re-raises) the exception that killed the task or `None` if that didn't happen. So the loop really should be

```

7 ...     for t in done:
8 ...         if t.exception() == None:
9 ...             print(t.result())

```

Tasks also have a `bool done` method: is it done yet? and a `get_coro` method that returns the coroutine that the task is built on. They also have a pair of methods `get_name` and `set_name` that allow you to give important tasks names so that you can recognise them. Names can be just about anything but `None`, not just strings. You can also name a task with an optional keyword `name` parameter to `a.create_task`.

`a.wait` has another keyword parameter called `timeout`. If provided, it must be a number, and specifies the longest time, in seconds, that the wait should last. It is not considered an error for the timeout to expire, the lists in the returned tuple will just reflect the state of affairs at the time of the expiration.

Now a slightly different situation. What if our URL list is not a list of requirements but a list of alternatives: successfully retrieving any one of the URLs is enough, we don't need them all. `a.wait` would still be exactly what we want, but after we get that first result, we need to stop all the other tasks. Unlike threads, tasks are very easy to stop. Just call their `cancel` method.

The next time a task gets its turn to run after having its `cancel` method called, an `a.CancelledError` exception will be raised. A task can capture `CancelledError` with a `try:` statement and make itself uncancelable. Tasks have a `bool cancelled` method, so you can tell what has happened. If a task that has created other (sub-)tasks gets cancelled, it is not entirely clear what happens. It is sometimes asserted that the sub-tasks get cancelled too, but experimentation shows that to be untrue. As far as I can tell, if the task was awaiting one of its sub-tasks at the time it got cancelled, that sub-task will also get cancelled, otherwise they seem to survive.

The `asyncio.shield` function can protect any awaitable from cancellation. After `aw2 = a.shield(aw1)`, `aw2` is an awaitable just like `aw1` except that it can't be cancelled. Not from outside anyway, a task can always cancel itself.

The `asyncio.wait` function is not the only way to wait until multiple tasks are done. There are a few alternatives. The first is the `gather` function. It can take any number of coroutines or tasks as parameters and returns a new awaitable. Awaiting that will wait until all are done, then return a list of all the tasks' final results, in the same order as the parameters. In the following, remember that if `coll` is some kind of collection object, and you pass `* coll` as a parameter, it gets replaced by all of `coll`'s contents as separate parameters. This is another alternative for `main`.

```
1 >>> async def main(URL_list):
2 ...     tasks = [ get_web_pg(u) for u in URL_list ]
3 ...     all = a.gather(* tasks)
4 ...     for r in await all:
5 ...         print(r)
```

The result of `gather` can itself be cancelled, and if it is, all of the things it is gathering (that is, its parameters) are also cancelled. If one of the gathered tasks has an uncaught exception then that exception will be raised by the `await all`. It is usually best to set `gather`'s optional keyword parameter `return_exceptions` to `True`. Then exceptions do not upset the `await`, they just appear in the list of results. As part of the following, I added `if random.random() < 0.5: raise ValueError()` to the `get_web_pg` coroutine in order to cause random exceptions, then changed `main` as follows. `repr` is used in the `print` to make exceptions visible.

```
1 >>> async def main(URL_list):
2 ...     tasks = [ get_web_pg(u) for u in URL_list ]
3 ...     all = a.gather(* tasks, return_exceptions = True)
4 ...     for r in await all:
5 ...         print(repr(r))
6         requesting www.abc.com
7         requesting xyz.com/data
8         requesting cde.org/x.html
9         'www.abc.com=> gn tmzbxvh iqnjulxzlaa prv'
10        ValueError()
11        'cde.org/x.html=>ozvs utyxgjubtifzvktwoejsuduz'
```

Another alternative is to use a task group. Task groups have their own `create_task` method, so new tasks can be created within them. You can not `await` a task group, but it does have an attribute called `_tasks` which is a set of the still-undone tasks that it contains, so you can watch that set shrinking, or give it to `a.wait`. Unfortunately task groups only work with a special form of the `with` statement. The `with` automatically awaits all the tasks when it reaches its end. As we need results from those tasks, we must clumsily create a list of all the tasks so their results can be seen later. If our tasks were totally self-contained,

meaning that they don't just get the URLs, they actually deal with the results as well, this would be much neater.

```
1 >>> async def main(URL_list):
2 ...     tasks = []
3 ...     async with a.TaskGroup() as grp:
4 ...         for url in URL_list:
5 ...             task = grp.create_task(get_web_pg(url))
6 ...             tasks.append(task)
7 ...         for t in tasks:
8 ...             print(t.result())
```

An `async` with `may` only be used on objects that have `__aenter__` and `__aexit__` methods, just like the `__enter__` and `__exit__` methods needed by a normal `with`. But these methods must be defined with an `async def` inside the class. A `TaskGroup`'s `__aenter__` just creates an empty group, its `__aexit__` awaits all of the tasks in the group. An `async wait`, and therefore a `TaskGroup` is not nearly so helpful as an ordinary `with`.

If any task in a task group experiences an uncaught exception, all the other tasks in that group get cancelled. This rule does not apply to `CancelledError`. If one of the tasks is cancelled, the others continue happily, but you must be careful not to try to look at anything from a task without first checking that its `cancelled()` returns `False`.

Tasks may be given callback functions. A callback is a function that is automatically called when some particular thing happens, and for tasks that thing is ending. If you say `t.add_done_callback(f)` then when the task `t` is done, `f(t)` will be called. You can also use `t.remove_done_callback(f)` if you change your mind. A task may have any number of callback functions. It is not easy to do very useful things this way, as tasks don't carry much usable data. You can change that though. `asyncio`'s `current_task` function returns the currently running task, and you can always use `setattr` to add new attributes to an object, so long as you are careful to avoid any attributes that already exist. So a coroutine could put something into its task object with `setattr(a.current_task(), "__data", x)` then the callback function `f(t)` can retrieve that data with `getattr(t, "__data")`.

There is also an `asyncio.all_tasks` method which returns a copy of the list of all currently un-done tasks. Being a copy, you can safely remove things from it without doing any harm. This means that you can design your main coroutine to reliably wait until all tasks are finished by getting all tasks, removing itself from the list, and then awaiting it. `main` needs to remove its own task because if you wait for yourself to finish, you are obviously never going to finish.

```
1 ...     all = a.all_tasks()
2 ...     all.remove(a.current_task())
3 ...     await a.wait(all)
```

Returning to the web spider, there is another tidy way of seeing all the results as soon as they are ready. `asyncio.as_completed` takes a list, set, etc of tasks or coroutines and returns an iterator. Tasks appear in the iterator only when they are done, so if you print things from the iterator you will be seeing them in order of their completion. The things in the iterator still have to be awaited. You can use a for loop on the iterator or repeatedly use `next`.

```
1 >>> async def main(URL_list):
2 ...     cos = [ get_web_pg(url) for url in URL_list ]
3 ...     iter = a.as_completed(cos)
4 ...     for r in iter:
5 ...         print(await r)
```

Many `asyncio` functions accept a `timeout` parameter, and there are a few ways of applying timeouts to other operations.

```
async with asyncio.timeout(5.1): await something
```

If the something takes more than 5.1 seconds it gets cancelled and a `TimeoutError` is raised here

```
async with asyncio.Timeout(t): await something
```

Note the capital `T` this time. This isn't particularly useful, the time specified (`t`) is not a duration, but an absolute time. It is like saying "time out at two o'clock" instead of "time out after two hours". The problem with this function is that its idea of the time doesn't correspond with anything much at all. It is still counted in seconds, but nobody knows what time corresponds with zero.

The only way to get a handle on it is to get hold of the event loop and ask it what it thinks the time is, but if you're going to do that, you might as well use the version with a little `t` instead. This gets the event loop's current time:

```
a.get_event_loop().time()
```

This function's one advantage is that you can change the timeout value. If you don't know what the timeout should be when you are setting it up, you can use `None` as the value, that means never time out. Later when you do know the time, you can use the `Timeout`'s `reschedule` method:

```
1 ...     async with asyncio.Timeout(None) as timo:
2 ...         ...
3 ...         timo.reschedule(whatever)
```

You can also use `bool timo.expired()` to see if it has already expired, and `timo.when()` to see the current timeout value.

```
aw2 = asyncio.wait_for(aw1, seconds)
```

The awaitable `aw2` is exactly the same as the awaitable `aw1`, except that it has a built-in self-destruct method. After the given number of seconds has expired, a `TimeoutError` is raised. The timeout is relative to when the `await` starts, not when `wait_for` was called.



The event loop can also be used to arrange for ordinary functions to be called, usually with a delay. This can only be done from within a task because elsewhere there is no event loop. After `el = a.get_event_loop()`,

```
el.call_soon_threadsafe(f, a, b, c)
```

`f` will be called with parameters `(a, b, c)` as soon as possible. There may be any number including zero of parameters. As soon as possible means as soon as the current task awaits something and gives the event loop a chance to do its job.

```
el.call_at(t, f, a, b, c)
```

`f` will be called with parameters `(a, b, c)` as soon as possible after absolute time `t` has been reached. Absolute time means the same as for the capital `T` Timeout.

Back to the web spider example again. Our `get_web_pg` coroutine fakes the network activity by sleeping for a little while. If we were doing it properly, we would face the fact that the nice simple way to use the network involves blocking calls, something like `recv`, that will wait until something arrives. Real blocking calls and coroutines just don't mix. In an earlier section we saw a lot of ways to make network activities and keyboard waits non-blocking, but they were all quite complex. `asyncio` gives us two more alternatives. The first is very simple, but also very unsatisfactory.

`asyncio.to_thread(f, a, b, c)` calls the function `f` with parameters `(a, b, c)` in a new thread and creates an awaitable around that. Being in a separate thread means that it doesn't matter that it blocks, all our coroutines are in our own thread so they won't be affected. `to_thread` returns a coroutine, not a task. Awaiting a coroutine builds a task around it, so you can do that, but if instead you give it to `a.create_task`, it can start work as soon as possible, and by the time you get around to awaiting that task, it might be done already.

```
1 ...     tsk = a.create_task(s.to_thread(slow, url))
2 ...         # do something else coroutinely for a while
3 ...     answer = await tsk
```

The bad thing is that threads are much heavier and more expensive than coroutines, so we wouldn't be able to use this trick many times.

## ii. Blocking functions as coroutines

If you have an ordinary function that blocks for a while (that is, it takes a long time to finish), you can still make it into a properly awaitable coroutine. It requires some work at a quite low level, but works quite easily.

There is a module called `concurrent.futures.thread` that defines a class called `ThreadPoolExecutor`. A `ThreadPoolExecutor` controls a number (`max_workers`) of threads that are set aside for running ordinary blocking functions as

coroutines. It doesn't matter that the functions block because they are in a different thread from all of the coroutines. `concurrent.futures` has a `ThreadPoolExecutor` function that calls the constructor without you having to type the `.thread.` part.

We also need to get hold of the event loop and tell it to run our blocking function in this `Executor`. The result is an ordinary task.

In this very basic example, the function `slow` represents our blocking function. It doesn't really do anything, it just simulates a long computation with a sleep. Notice that it uses `time.sleep`, not `asyncio.sleep`, so it really is not compatible with coroutines. We also have an ordinary coroutine called `dotty`. It just prints a series of dots with one second delays. The point of `dotty` is that its trail of dots shows that coroutines can run along with our blocking function.

```
1 >>> import asyncio as a
2 >>> import concurrent.futures as cf
3 >>> import time
4
5 >>> def slow():
6 ...     time.sleep(10)
7 ...     return "done"
8
9 >>>
10 ... async def dotty():
11 ...     while True:
12 ...         print(".", end = "")
13 ...         await a.sleep(1)
```

Now for the main coroutine. Just six lines. The first creates and starts an ordinary task based on `dotty`, `td = task dotty`. The second gets hold of the event loop. The third creates the `ThreadPoolExecutor`, The fourth tells the event loop to make `slow` into a task using that executor, `ts = task slow`. Finally we `await` task `slow` and show its result. What we see is a trail of ten dots (`slow` sleeps ten seconds) slowly appearing across the screen until suddenly everything is over. There is no `await` on task `dotty`, so it doesn't keep the main coroutine alive.

```
1 >>> async def main():
2 ...     td = a.create_task(dotty())
3 ...     loop = a.get_event_loop()
4 ...     exec = cf.ThreadPoolExecutor(max_workers = 3)
5 ...     ts = loop.run_in_executor(exec, slow)
6 ...     await ts
7 ...     print("the result was", repr(ts.result()))
8
9 >>>
10 ... a.run(main())
11 .....the result was 'done'
```

`asyncio` provides a nice simple way of making a coroutine wait until keyboard input is ready. Unfortunately, as always, it doesn't work under Windows. But we can combine this new technique with the `pynput` module covered in the section on

polling. This makes use of the keystroke decoding function `decode` from that section, so I won't show it again here, and also an ordinary `queue.Queue`.

The `run` function sets up a `pynput` keyboard listener with a function down that adds keystrokes to a queue as they are received, and starts the main coroutine. The call to `a.run(main())` really should be in a `try:` so we can be sure the Listener will be stopped properly.

```
1 >>> def down(key):
2 ...     global q
3 ...     q.put(key)
4
5 >>> def up(key):
6 ...     pass
7
8 >>> def run():
9 ...     L = kb.Listener(on_press = down, on_release = up)
10 ...     L.start()
11 ...     a.run(main())
12 ...     L.stop()
```

We also use the `dotty` coroutine from above, and another, `sleepy`, just for a bit of variety. The main coroutine starts `sleepy` and `dotty` tasks, and a special task for the blocking function `getstring`. We wait for both `sleepy` and `getstring` to finish.

```
1 >>> async def sleepy():
2 ...     await a.sleep(7)
3
4 >>> async def main():
5 ...     ts = a.create_task(sleepy())
6 ...     td = a.create_task(dotty())
7 ...     loop = a.get_event_loop()
8 ...     exec = cf.ThreadPoolExecutor(max_workers = 3)
9 ...     tg = loop.run_in_executor(exec, getstring)
10 ...     await a.wait([ts, tg])
11 ...     print("it was", repr(tg.result()))
```

`getstring` just absorbs the keystrokes that `down` puts into the queue, decodes them into proper characters and builds up a string from them. As soon as `enter` is pressed, it returns the string that was typed.

```
1 >>> def getstring():
2 ...     global q
3 ...     s = ""
4 ...     while True:
5 ...         c = q.get()
6 ...         if c == kb.Key.enter and s != "":
7 ...             print("the string is", repr(s))
8 ...             return s
9 ...         s += decode(c)
10
```

```

11 >>>
12     run()
13     .....the string is 'hello'
14 >>> it was 'hello'
15     run()
16     ..the string is 'hello'
        .....it was 'hello'

```

In the first run I typed `hello` very slowly, and as soon as I pressed enter both replies appeared and the run was over. In the second run I typed quickly. When I pressed enter we immediately saw the message from `getstring`, the string is `'hello'`. Then there was a wait of five more seconds because `sleepy` hadn't finished yet. As soon as `sleepy`'s seven seconds were up, `main`'s `await` ended and we saw the second message.

### iii. TCP clients

A TCP client takes no more effort than it does with normal functions. `asyncio` provides a coroutine for the purpose:

`a.open_connection(host, port)` - an awaitable coroutine

As with the normal networking function `connect`, `host` is a host name or IP address, and `port` is the desired port number. `await` returns a two-tuple containing a `StreamReader` object and a `StreamWriter` object. Those objects behave mostly as files, and they are how you communicated with the server.

Optional parameters include `limit` (default 65536) the maximum buffer size used by the `StreamReader`, and `ssl` (default `False`): should this be a secure connection or an ordinary one. For example, if you are using HTTP then `ssl` should be `False` and `port` should be 80. If you are using HTTPS then `ssl` should be `True` and `port` should be 443.

`StreamReaders` provide

`readline()` - an awaitable coroutine

Read exactly one line and return it as a `bytes` object. The `\n` or `\r\n` that ended the line will be part of the result. A zero-length `bytes` object will be returned if the connection has been closed and nothing remains to be received.

`read(n)` - an awaitable coroutine

Read `n` bytes and return them as a `bytes` object. You may receive fewer than `n` bytes. A zero-length `bytes` object will be returned when the connection has been closed and nothing remains to be received.

`readuntil(sep)` - an awaitable coroutine

The same as `readline` except that `sep`, a `bytes` object, will be used as the end-of-line character instead of the usual `\n`. Actually there is another difference. If there is no more input, instead of returning an empty object it will raise an `IncompleteReadError` exception.

`readexactly(n)` - an awaitable coroutine  
Exactly the same as `read`, except that an `IncompleteReadError` exception will be raised if fewer than `n` bytes are available. The exception's `partial` attribute is a `bytes` object that will contain those bytes that were received.

`StreamWriters` provide

`write(bts)` - an ordinary function, do not await it  
`bts`, a `bytes` object is prepared for transmission. If it can be sent immediately without blocking it will be. If it can't, it will be kept safe until it can be sent.

`writelines(it)` - an ordinary function, do not await it  
`it`, an iterable of `bytes` objects, are all prepared for transmission as for `write`.

`drain()` - an awaitable coroutine  
`await w.drain()` waits until all data previously prepared for transmission has been sent.

`close()` - an ordinary function, do not await it  
Starts the procedure closing both `Stream` objects and the connection itself. Should be followed by `wait_closed()`

`wait_closed()` - an awaitable coroutine  
`await w.wait_closed()` waits until the connection is fully closed.

Here is the simple web client from the networking section converted to work as a coroutine:

```
1 >>> import asyncio as a
2 >>> async def ask():
3 ...     host = "rabbit.eng.miami.edu"
4 ...     (rdr, wrtr) = await a.open_connection(host, 80)
5 ...     wrtr.write(b"GET /findme.html HTTP/1.0\r\n")
6 ...     wrtr.write(b"\r\n")
7 ...     await wrtr.drain()
8 ...     while True:
9 ...         bts = await rdr.readline()
10 ...         if len(bts) == 0:
11 ...             break
12 ...         print(bts.decode())
13 ...         wrtr.close()
14 >>> await wrtr.wait_closed()
15 >>> a.run(ask())
16
```

...

#### iv. TCP servers

Servers are naturally a little more complex. We'll go back to the very basic time server from the networking section. It will wait for a connection, read one line from it, ignore that line, and send back the current time.

When a server is started, it must be provided with a callback coroutine. Every time a connection is accepted, that coroutine will be run with the `StreamReader` and `StreamWriter` for the connection as parameters.

It makes sense, but is certainly not required, to implement a server as a class. We'll start off with just the callback coroutine.

```
1 >>> import asyncio as a
2 >>> import time
3
4 >>> class timeserver:
5 ...
6 ...     async def handle_connection(self, rdr, wrtr):
7 ...         print("from", wrtr.get_extra_info("peername"))
8 ...         bts = await rdr.readline()
9 ...         print("received", bts.decode())
10 ...        wrtr.write(time.strftime("%c").encode())
11 ...        wrtr.write(b"\r\n\r\n")
12 ...        await wrtr.drain()
13 ...        wrtr.close()
14 ...        await wrtr.wait_closed()
```

There is only one new thing here, the first line. After that, the callback coroutine for a server does exactly what a client does. Without the `open_connection` of course.

`get_extra_info` is only available for `StreamWriters`, and it is not very well documented at all. `"peername"` is a request for information on the client that connected, it is a tuple of IP address and port number.

Now we can safely move on to the main coroutine. It just starts a server and tells us which port number to use to connect to it.

```
1 ...     async def main(self):
2 ...         s = await a.start_server(self.handle_connection, "", 0)
3 ...         print("port", s.sockets[0].getsockname()[1])
4 ...         async with s:
5 ...             await s.serve_forever()
```

The `start_server` method does all the socket and bind and listen business automatically. Its second and third parameters are our IP address where the

empty string is taken to mean the local host "127.0.0.1", and the desired port number with zero meaning "just pick one for me".

The next line looks a bit unpleasant, but is the only way I could find for discovering a randomly assigned port number. While the server is running, its `sockets` attribute is a list of all the sockets it is using. Before accepting any connections there will only be one. Its `getsockname` method returns a list of IP address and port number.

The `async` with ensures that the server will be correctly shut down even if an exception occurs. `serve_forever` just means carry on until you are stopped. It contains the loop with the `accept` call and calls the callback coroutin every time `accept` yields a result. All we need now is a constructor to start it all.

```
1 ...     def __init__(self):
2 ...     a.run(self.start())
3
4 >>> timeserver()
5     ...
```

## v. UDP agents

There is of course also support for UDP within `asyncio`. It seems to be less evolved, we have to use lower level objects like the event loop directly. But it isn't too bad.

First we have to get hold of the event loop, then ask it to create the UDP socket, which it calls a `datagram` endpoint. The `create_datagram_endpoint` method must be given two parameters, a zero-parameter function that returns an object to handle communications, and a `local_addr` which is a tuple containing the usual IP address (local host) and desired port number. You then wait for as long as you want the agent to run, and then close the port.

The object provided to handle communications should have these four methods. You can get away without the last two, but it is fairly important to be able to detect errors at least. These methods are not coroutines, just ordinary `defs`:

```
connection_made(self, transp):
```

This is called just once, when the port is first set up. It makes the handler object aware of the transport to be used. A transport is just an object with useful methods such as `sendto` and `close` for communicating with another agent.

Remember that UDP does not deal with connections. This, and its friend `connection_lost`, are rather badly named. They are only to do with setting up and closing the socket.

```
datagram_received(self, message, sender):
```

This is called every time anything is received. `message` will be a `bytes` object, the data that was received, and `sender` will be a tuple of IP address and port number.

```
connection_lost(self, x):
```

This is called just once when the UDP port is shut down. It must have one parameter, but it never seems to be used.

```
error_received(self, x):
```

This is called when the system detects that a send operation failed. This very rarely happens.

Here is something very minimal:

```
1 >>> class udp_controller:
2 ...
3 ...     def connection_made(self, transp):
4 ...         self.transp = transp
5 ...
6 ...     def datagram_received(self, message, sender):
7 ...         print("From", sender, message)
8 ...
9 ...
10 ...     def error_received(self, x):
11 ...         print("error received", type(x), x)
12 ...
13 ...     def connection_lost(self, x):
14 ...         pass
15 ...
16 ...     def send(self, message, where):
17 ...         self.transp.sendto(message.encode(), where)
18 >>>
19 ...     def maker():
20 ...         return udp_controller()
```

The last function, `maker`, is the one we will give to `create_datagram_endpoint` for obtaining the handler object. When the socket is created, the `connection_made` method is called with a transport parameter set to that object.

To illustrate all of this, we'll make a program that creates a UDP port and makes its number known. Then it just waits. If a datagram is received it, along with its sender, is displayed. If a whole line is typed at the keyboard, it is expected to have this format: `host:port message`. `host:port` would be something like `127.0.0.1:5642`, and `message` is obvious. To make testing easier, we'll make the hostname optional, defaulting to `127.0.0.1`, which refers to the computer's "loopback" interface, it just means "this computer". Dealing with the keyboard is very system dependent, I'll go with a Windows version, using things we developed in the `pynput` subsection of the section on polling, and I won't repeat that stuff here. Under Unix, the standard Python function `input()` can be made into a task. Under Windows, no surprise, it can't.



There is quite a large preamble:

```
1 >>> import asyncio as a
2 >>> import concurrent.futures as cf
3 >>> from pynput.keyboard._win32 import KeyCode
4 >>> from pynput import keyboard as kb
5 >>> from queue import Queue
6 >>> import win32.win32gui as wg
7 >>> import socket
8
9 >>>
10 >>> kbq = Queue()
11 >>> hname = None
12 >>> realip = None
    mywin = wg.GetForegroundWindow()
```

The queue is there so that the keyboard handling `down` function can send the keypresses wherever they are needed. `hname` and `realip` are to store this computer's real name and IP address.

To make this illustrate all the possibilities, there will be three extra features: If the user types just `stop` at the keyboard, the system will close down. If the message received over UDP is `reply`, we'll automatically send back a reply. If you send another agent a `stop` message, it will close down.

The `datagram_received` method from above gets a bit bigger. `message` will be a bytes object (remember that `.decode()` turns it into a string), and `sender` will be the usual (host, port) tuple.

```
1 ...     def datagram_received(self, message, sender):
2 ...         message = message.decode()
3 ...         self.queue.put((sender, message))
```

The controller object will have its own queue, just like `kbq` from above, that will store all received messages until they are dealt with. That means we need a constructor to create that queue and a get method to give access to it.

```
1 ...     def __init__():
2 ...         self.queue = Queue()
3 ...
4 ...     def get(self):
5 ...         return self.queue.get()
```

We will also need a function that accumulates a whole line of keypresses into a string. A bit larger than our earlier `getstring` so that it can at least handle backspaces.

```
1 >>> def getstring():
2 ...     global kbq
3 ...     s = ""
4 ...     while True:
5 ...
```

```

6 ...     c = decode(kbq.get())
7 ...     if c == "\n":
8 ...         return s
9 ...     if c == "\x08" and s != "":
10 ...         s = s[: -1]
11 ...     else:
12 ...         s += c

```

The `get` method and the `getstring` function are blocking functions that will be turned into tasks by the `run_in_executor` function. So our main coroutine will be like this.

```

1 >>> async def main():
2 ...     global hname, realip
3 ...     eloop = a.get_event_loop()
4 ...     transp, ctrl = await eloop.create_datagram_endpoint(
5 ...         maker,
6 ...         local_addr = ("127.0.0.1", 0))
7 ...     print("waiting on port")
8 ...     transp.get_extra_info("sockname")[1])
9 ...     loop = a.get_event_loop()
10 ...     exec = cf.ThreadPoolExecutor(max_workers = 6)
11 ...     tkb = loop.run_in_executor(exec, getstring)
12 ...     tnet = loop.run_in_executor(exec, ctrl.get)
13 ...     while True:
14 ...         ...
15 ...         transp.close()

```

`create_datagram_endpoint` is given, as well as the handler making function, a local address of `("127.0.0.1", 0)`. This can be used to request a specific port, but as before, zero asks the system to just find a free one. It returns a tuple of a transport and a protocol. The protocol is exactly the controller object that `maker` created for it, we need that for sending messages.

On entering the `while` loop then, we have two tasks already running. We need to wait until either one of them delivers a result. If it was `tkb`, the keyboard task, we need to process what was typed and deal with it. If it was `tnet`, the network task, we need to process the message and continue.

```

13 ...     while True:
14 ...         (done, undone) = await a.wait([tkb, tnet],
15 ...             return_when = a.FIRST_COMPLETED)
16 ...
17 ...         if tkb in done:
18 ...             ok = handle_keyboard(tkb, ctrl)
19 ...             if not ok:
20 ...                 break
21 ...             tkb = loop.run_in_executor(exec, getstring)
22 ...
23 ...         if tnet in done:
24 ...
25 ...

```

```

26 ...         ok = handle_network(tnet, ctrl)
27 ...         if not ok:
...             break
28         tnet = loop.run_in_executor(exec, ctrl.get)
        transp.close()

```

The `ok` variable is because a `stop` message from either source needs to be able to stop the loop. Notice that after each section, we have to recreate the task that gave its result. Once a task has delivered its result to an `await`, it is done. A finished task can't be restarted, so we need to create a new one. Each time round the loop, the `await a.wait` will have one old task and one new one to wait for.

The keyboard event processing that was elided in that isn't very complicated. Check for `"stop"`. Split what was typed into `host`, `port`, and `message`. Send the message. The splitting function won't be shown, it is just basic string processing. A `try` is needed so that the parsing function can conveniently signal format errors.

```

1 >>> def handle_keyboard(tkb, ctrl):
2 ...     msg = tkb.result()
3 ...     print("from keyboard: \", msg, "\", sep = ")
4 ...     if msg == "stop":
5 ...         return False
6 ...     try:
7 ...         (host, port, msg) = parse_msg(msg)
8 ...     except:
9 ...         print("bad format")
10 ...     else:
11 ...         ctrl.send(msg, (host, port))
12 ...     return True

```

And the handler for a network message isn't much different:

```

13 >>> def handle_network(tnet, ctrl):
14 ...     (sender, msg) = tnet.result()
15 ...     if msg == "reply":
16 ...         ctrl.send(b"Here I am.\n", sender)
17 ...     else:
18 ...         if msg == "stop":
19 ...             return False
20 ...         print("from ", sender, ": ", msg, sep = ")
21 ...     return True

```

To bring everything together we need a few more things. Under Unix, the `sendto` method accepts host names in both numeric `"129.171.33.6"` and human `"acme.com"` formats. Under Windows it only accepts the numeric format. To fix that, we'll have the following little function. It isn't perfect, it assumes that anything that begins with a digit will be in the numeric format, which is not correct.

```

1 >>> def getip(s):

```

```

2 ...     if s != "" and s[0].isdecimal():
3 ...         return s
4 ...     try:
5 ...         return socket.gethostbyname(s)
6 ...     except:
7 ...         return ""

```

And to find our own host name and IP address, this is all that's needed.

```

1 >>> def findself():
2 ...     global hname, realip
3 ...     hname = socket.gethostname()
4 ...     realip = socket.gethostbyname(hname)

```

Now for the function that starts everything. Find our address, set up the keyboard listener, and start the main coroutine.

```

1 >>> def run():
2 ...     L = kb.Listener(on_press = down, on_release = up)
3 ...     L.start()
4 ...     try:
5 ...         a.run(main())
6 ...     except:
7 ...         pass
8 ...     L.stop()

```

The `try` is needed to ensure that the keyboard listener is stopped even if something goes wrong. That isn't really satisfactory. If something does go wrong, your program will just stop silently and you will have no idea why. It would be better to see the exception and the usual trace-back information. Just import `traceback` and replace the `try: pass` with this:

```

6 ...     except BaseException as e:
7 ...         print("Exception in run():")
8 ...         traceback.print_exception(e)
9 ...     L.stop()

```

There is one further concern. When you are experimenting on a single computer, it makes sense to use the loopback interface `127.0.0.1`, but for real life use, you would want to use real IP addresses. Even though they are sockets on the same computer, a socket that was given one of those as its local address will not be able to communicate with any sockets on the other. To make this work, you would need two controller objects and two tasks:

```

1 ... transpl, ctrl1 = await eloop.create_datagram_endpoint(
2 ...     maker,
3 ...     local_addr = ("127.0.0.1", 0))
4 ... print("waiting on localhost/127.0.0.1:",
5 ...     transpl.get_extra_info("sockname")[1], sep = "")
6 ... transpp, ctrlp = await eloop.create_datagram_endpoint(
7 ...     maker,
8 ...

```

```

9 ...             local_addr = (realip, 0))
10 ... print("waiting on ", hname, "/", realip, ":",
11 ...       transpp.get_extra_info("sockname")[1], sep = "")
12 ...
13 ... tnetl = loop.run_in_executor(exec, ctrl1.get)
13 ... tnetp = loop.run_in_executor(exec, ctrlp.get)

```

The names that end in “l” are for loopback addresses, and those that end in “p” are for public addresses. And of course `a.wait` would need to be given all three tasks `[tkb, tnetl, tnetp]`. It also means that there needs to be a separate `if` for the two `tnet` tasks. The incompatibility between the two types of sockets means that `handle_keyboard` needs to be careful to transmit messages through the correct controller:

```

1 >>> def handle_keyboard(tkb, ctrl):
2 ...     ...
3 ...     try:
4 ...         (host, port, msg) = parse_msg(msg)
5 ...     except:
6 ...         print("bad format")
7 ...     elif host == "127.0.0.1":
8 ...         ctrl1.send(msg, (host, port))
9 ...     else:
10 ...         ctrlp.send(msg, (host, port))
11 ...     return True

```

This large example encountered many ways in which Unix and Windows behaviours are completely incompatible. To get around that, just remember what we did in the system independence subsection of the operating system features section.

## vi. Other protocols

Unfortunately `asyncio` hasn’t gone very far with the more user-oriented protocols. There is no built-in support for web services, HTTP, of the sort we saw in the network clients and servers section. There are some third party packages available, but I haven’t seen enough about them to be able to trust them to be safe.

You will probably just have to implement any protocol you need by yourself.

# Part Two: Graphics and User Interfaces

If all you need is to produce a graphical display, perhaps to show a picture or a graph, the task is exceptionally easy. Things only get complicated if you want the user to be able to interact in some way. There are two approaches:

`Turtle` is the simplest, it allows you to control an imaginary pen moving across an imaginary piece of paper. The paper isn't entirely imaginary, it is in fact a window that automatically appears on the screen. The pen can change its colour and width and move in all sorts of ways. The idea of the turtle as a driver of graphics is quite old, from the early 1960's, and was aimed entirely at children. It was thought that a turtle moving around would be more child-friendly, and at the beginning it really did have a robot turtle with a real pen attached to it.

The other approach, `tkinter`, allows the creation of complete graphical user interfaces with buttons, text entries, graphical displays, and all of the other things that we are used to. With `tkinter`, simple things are indeed quite simple to do. The programming only gets complicated when you need to do complicated things.

## 38. Turtle

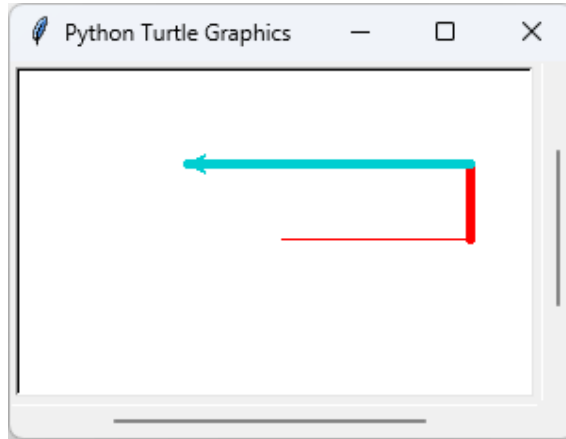
First, BEWARE: The error messages, or exceptions, produced by `turtle` are appallingly bad. They provide no clue about what went wrong.

To make a quick start, just issue one of the `turtle` commands and a window pops up instantly, showing the results. The default window size is quite big, which is usually good, but here it is bad for including screenshots so I'll start by setting the pen colour. That counts as a `turtle` command, so a window springs into existence. Then the `setup` method can be used to set the width and height. The turtle itself appears as an arrowhead.

Using Idle on my Windows PC, the first time I enter a command to make the window appear, it works. If I close the window and try to create a new one, it always fails. But if I do the same thing for a third time, it works again. It alternates between failing and succeeding with each attempt. Sometimes, at seemingly random points, `turtle` just goes mad and crashes my Idle session. Occasionally the turtle doesn't initially appear. `tu.st()` will fix that.

```
1 >>> import turtle as tu
2
3 >>> tu.color("red")
4 >>> tu.setup(300, 200)
5 >>> tu.forward(100)
6 >>> tu.left(90)
7 >>> tu.width(5)
8 >>> tu.forward(40)
9 >>> tu.color("darkturquoise")
10 >>> tu.left(90)
```

```
11 >>> tu.forward(150)
```



`setup(width, height, x, y)` changes the size of the visible window, not the size of the underlying imaginary sheet of paper. `x` and `y` are optional, they set the position of the window's top left corner relative to the screen's top left corner. Scrollbars automatically appear if the window is not at least a little bit bigger than the paper. `screensize(width, height)` sets the size of the imaginary piece of paper, but it is not very useful. If you draw off the edge of the paper, the drawing still happens and is revealed if the window is stretched or the scrollbars are moved. The only real use for `screensize` is that it is used to determine the size of the scroll bars. If you draw over the edge, the scroll bars might not be able to reveal everything. The current dimensions of the window are returned by `window_width()` and `window_height()`. `title(string)` changes the text in the window's title bar.

`forward(n)` means move in the current direction by `n` pixels. At startup, the turtle is always at the centre of the window, pointing East. The turtle maintains knowledge of its current direction as it moves and turns.

`left(n)` and `right(n)` mean change direction by `n` degrees. `heading()` returns the current direction. `backward(n)` makes the turtle move in the opposite direction from the way that `forward` would have moved it, but it does not change the turtle's idea of its current direction. The `degrees(d)` method changes the way directions and angles are measured. The default value is 360 meaning things are measured in degrees. If `d` is given as  $2 \times \pi$  measurements will be in radians. But `d` can be any value, it specifies how a turn by a full circle would be represented, so if `d` is 1, 0.25 would mean a right angle. The `radians()` method is equivalent to `degrees(2 * math.pi)`.

`forward`, `backward`, `left`, and `right` may be abbreviated to `fd`, `bk`, `lt`, and `rt`.

`width(sz)` sets the width of the line that the pen draws, in pixels. It may also be written as `pensize(sz)`.

`color()` sets the colour of the line that is drawn when the pen is down. Python knows an enormous number of colour names, but there doesn't seem to be any

Python way of finding out what they are. A web search easily finds a long list of names and samples of what they look like. Alternatively you can use a three-tuple (R, G, B) to specify a colour. R, G, and B are the red, green, and blue colour components in the range 0 to 1 (floats are obviously allowed), but you may change the range, `colormode(255)` will change it to 0 to 255, and `colormode(1)` will change it back again. `colormode()` with no parameters returns the current setting. Another alternative is to express a colour as a string in this format "#RRGGBB", where R, G, and B are hexadecimal digits giving the colour components in the 0 to 255 range.

The `color` method may be given two colour parameters, in which case the first sets the pen colour, and the second sets the colour to be used when a shape is filled in. `color()` with no parameters returns a tuple of two three-tuples which are the current pen and fill colours, expressed according to the current `colormode`. two extra methods, `pencolor` and `fillcolor`, may also be used. They set or return just a single one of the pen or fill colours. `bgcolor(colour)` sets the background to the drawing. It does not cover up the drawing, it just slides in beneath everything. `bgcolor()` returns the current background colour. Alternatively, if you have an graphical image that you would like to use as the background, this is the way to do it: `bgpic("images\\background.png")`. Not many file formats are supported, but png and gif always seem to be.

The pen can be either up or down, up allowing it to be moved without making any marks, just changing its position. The `penup` and `pendown` parameterless methods do that job, and they may be abbreviated to `pu` or `up` and `pd` or `down`. `isdown()` returns `True` or `False` so that you don't have to record the pen's state. If the pen is up, putting it down then up again with no intermediate movement does not leave a mark.

Things may also be changed in absolute terms instead of the relative terms used by `forward` and `left`. `setpos(x, y)` or `setpos((x, y))` move the turtle to the given coordinates, measured in pixels. Unlike most computer graphics systems, the turtle uses normal mathematical conventions: larger y values are up, smaller ones are down, larger x values are right, smaller ones are left, and (0, 0) is the centre of the window. if the pen is down, `setpos` will draw a straight line, if it is up, it won't. `setposition` and `goto` are both equivalent to `setpos`. `setx(x)` and `sety(y)` just change one dimension of the turtle's position, leaving the other unchanged. Setting the turtle's position does not affect its heading. `pos()` with no parameters, it can also be called `position()`, returns an (x, y) tuple giving the turtle's current position. `home()` returns the turtle to (0, 0) and its original heading. `xcor()` and `ycor()` return the single x or y coordinate for the pen's position.

`setheading(a)` or `seth(a)` sets the current direction in absolute terms. a being zero means right or East, and increasing angles move anti-clockwise, so a being 90° means up or North. But these conventions can be changed by setting the mode.



These are a few useful methods for tidying things up. `clear()` clears the screen but leaves the turtle unaffected. `reset()` clears the screen and sets everything back to how it was at startup. `undo()` is what the name suggests, everything goes back to the way it was before the last command, except that `reset()` can't be undone. Commands are kept in a stack, calling `undo()` four times will undo the last four commands, but the stack size is limited. `setundobuffer(n)` sets the maximum size of the undo stack to `n` commands, but it also empties the stack. `setundobuffer(None)` completely disables `undo()`. `undobufferentries()` returns the number of commands currently on the undo stack.

The `mode()` method returns the current mode, it will be one of "standard", "logo", or "world", "standard" is the default. `mode(s)`, where `s` is one of those strings, changes the current mode. "standard" is as described above. "logo" means that direction 0 is up or North, and increasing angles move clockwise, so 90° means right or East. "world" is for use with the `setworldcoordinates` method, coming up soon.

The appearance of the turtle may be changed and monitored: `hideturtle()` or `ht()` makes it invisible, but it still behaves exactly as normal. `showturtle()` or `st()` makes it visible again. `isvisible()` returns `True` or `False`. `shape()` returns a string indicating how the turtle appears, it will be one of "arrow", "turtle", "circle", "square", "triangle", or "classic", but the `getshapes()` method returns a list of all registered shapes, so it would be better to use that in case there is something system-dependent about the list. `shape(s)`, where `s` is one of those strings, sets the turtle's shape accordingly. These six screenshots show the different shapes in the order given here. They have been enlarged.



The turtle's size may be controlled in one of two ways, as selected by the `resizemode` method. It takes one string parameter. "noresize" means the turtle's size will never change. "auto" means the turtle will be resized whenever the pen width is changed, in order to keep it in proper proportion. "user" means that the programmer will set the turtle's size with the `turtlesize` method. On my Windows PC "noresize" has no special effect, it is exactly the same as "user".

`turtlesize(stretch_wid = None, stretch_len = None, outline = None)` may also be written as `shapesize`. Any parameter left as `None` causes no change. `stretch_wid` and `stretch_len` are not the new sizes, they are what the turtle's natural size should be multiplied by. `outline` is a number of pixels: the turtle is always drawn as a shape in the fill colour with an outline in the normal pen colour. With no parameters, `turtlesize()` returns a three-tuple of the current settings.

There are some very obscure methods that allow you to turn the turtle to an angle that does not represent its heading and to distort its shape in pointless ways. I'll list them just for reference, but won't say anything more about them. `shearfactor`, `tilt`, `settiltangle`, `tiltangle`, `shapetransform`, `get_shapepoly`.

The `pen` method allows you to set a lot of the turtle's options at once, or to see all of them at once. `pen()` returns a dictionary of all the pen's current settings. The keys are: `shown` (True or False - is the turtle visible), `pendown` (True or False), `pencolor` and `fillcolor` (both colour strings), `pensize`, `outline`, and `speed` (all numbers), `resizemode` (a string), and `stretchfactor` and `tilt` (the obscure stuff). `pen(d)`, where `d` is a dictionary of exactly the same kind, sets all the settings that it mentions, if a key is missing, that setting is not changed. And also you can provide the settings individually as keyword parameters, such as `pen(pensize = 3, outline = 2)`.

A few more drawing methods: `circle(radius)` draws a circle starting at the turtle position and progressing by making a small step forward and a small turn to the left until the whole circle has been made. `circle(radius, extent)` where `extent` is an angle, draws an incomplete circle (an arc), it stops when the total of all the small turns adds up to `extent`. The turtle will end up at the end of the arc, pointing in the direction of the last small step forward. There is one further variation, `circle(radius, steps = s)` or `circle(radius, extent, s)` sets the size of the small steps forward so that an entire circle would be drawn in exactly `s` equally sized steps. Setting `s` to 7 for instance draws a regular heptagon (or part of one if `extent` is given) that would be inscribed inside the circle that would have appeared if the number of steps had not been specified.

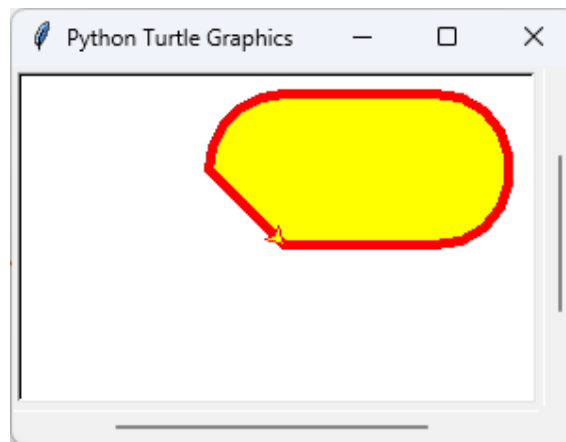
`dot()` draws a small dot of a size designed to still be easily seen. The diameter is either the pen width plus four, or twice the pen width, whichever is largest. The dot is drawn even if the pen is up. `dot(size)` does the same, but the diameter of the dot is the given number of pixels. On my Windows PC, a size of 1 is invisible. The colour of the dot may be set in a variety of ways: `dot(size, R, G, B)`, `dot(size, (R, G, B))`, `dot(size, string)`, `dot(None, R, G, B)`, `dot(None, (R, G, B))`, or `dot(None, string)`, the string being like "red" or "#FF0000".

`stamp()` makes an exact image of the turtle at its current position. After the turtle moves, the image remains in place until it is deliberately removed. `x = stamp()` captures an identifying number for the image, a later call to `clearstamp(x)` will remove it. `clearstamps(n = None)`, note the extra `s` at the end of the name. If `n` is `None`, all stamps are removed. If `n` is positive, e.g. 3, the oldest 3 surviving stamps are removed. If `n` is negative, e.g. -3, the newest 3 surviving stamps are removed.

Filling shapes is easy. First say `begin_fill()`, then move the turtle around the shape, it doesn't matter whether the pen is up or down, just move around the outline, then say `end_fill()`. Whatever shape you made gets filled in with the fill colour. It doesn't even matter whether you close up the shape or not, a straight line boundary from the `end_fill` position to the `begin_fill` position is

automatically added. If your shape's edges cross each other, the effect of filling is system dependent, but usually something a reasonable person would accept.

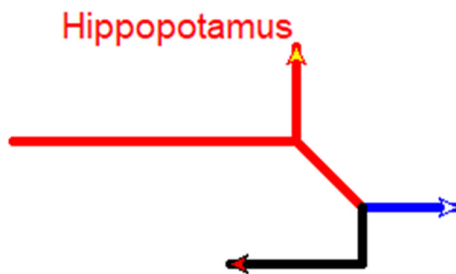
```
1 >>> tu.color("red", "yellow")
2 >>> tu.setup(300, 200)
3 >>> tu.begin_fill()
4 >>> tu.fd(80)
5 >>> tu.circle(40, 180)
6 >>> tu.fd(80)
7 >>> tu.circle(40, 90)
8 >>> tu.left(90)
9 >>> tu.right(45)
10 >>> tu.fd(math.sqrt(2 * math.pow(40, 2)))
11 >>> tu.end_fill()
```



Beware of one little thing. At least on my Windows PC, `undo()`ing an `end_fill()` doesn't work. The corresponding `begin_fill()` is ruined and can not be ended again.

You can have a number of turtles moving about at the same time. `tu.clone()` returns an absolutely identical copy of the turtle `tu`, but despite starting out identical, they now move and can be changed independently.

```
1 >>> import turtle as tu
2
3 >>> tu.color("red", "yellow")
4 >>> tu.width(5)
5 >>> tu.fd(150)
6 >>> tu2 = tu.clone()
7 >>> tu.lt(90)
8 >>> tu.fd(50)
9 >>> tu.write("Hippopotamus",
10 ...         align = "right",
11 ...         font = ("Arial", 14))
12 >>> tu2.rt(45)
13 >>> tu2.fd(50)
14 >>> tu2.color("blue", "white")
15 >>> tu3 = tu2.clone()
16 >>> tu3.color("black", "red")
17 >>> tu2.lt(45)
18 >>> tu2.fd(50)
19 >>> tu3.rt(45)
20 >>> tu3.fd(30)
21 >>> tu3.rt(90)
22 >>> tu3.fd(70)
```



There is a little oddity here. `tu` is the module that we imported, and the object returned from `tu.clone()` is not, it is of type `turtle.Turtle`, but almost all of the same methods work in the same way on both of them. `tu.turtles()` returns a tuple of all the turtles in the window. This is an example of a method that does not work on a `turtle` object, only on the module.

Only the first parameter to `write` is necessary, it can be anything with a `__str__` method to turn it into a string. The optional parameters are `move = True` or `False`, the default is `False`, if `True` the turtle moves to the bottom right corner of the text, if `False` it stays where it is. `align = "left", "right",` or `"center"`, the default is `"left"`, it says where the turtle is relative to the text. `font` can be a one-tuple (`"name",` ), a two-tuple (`"name", size`), or a three-tuple (`"name", size, "normal" or "bold" or "italic"`) where of course name is an existing font name.

Some useful navigational information is available too. `tu.towards(other)` returns the angle `tu` would have to turn left in order to be heading towards `other`, which must be another turtle object. `tu.towards(x, y)` and `tu.towards((x, y))` return the angle `tu` would have to turn left be in order to be pointing directly to that position. A similar method, `tu.distance` takes exactly the same parameter possibilities, and returns how far `tu` would have to move forward to reach that position if it were already pointing the right way.

`tu.speed(n)` determines how quickly lines are drawn. 0 is the fastest, but then there is a sliding scale of floats from 1 to 10 where 1 is the slowest, and 10 is just a little slower than 0. With no parameters it returns the current speed setting.

`tu.bye()` closes the window. `tu.exitonclick()` is something I can't imagine any use for. After calling it, a mouse click anywhere on the window will close it.

`setworldcoordinates(left, bottom, right, top)` changes the coordinate system used in all operations. `left` specifies the x value that the left edge of the window corresponds to, and the other three parameters have the obvious meanings in view of that. `setworldcoordinates(0, 0, 1, 1)` would result in (0, 0) being the bottom left corner and (1, 1) being the top right, all x and y values would be floats between 0 and 1. If the parameters are "in the wrong order", you can turn the coordinate system upside-down to match most other graphics systems, e.g. ... (0, 500, 500, 0). If the mode is already "world", the screen is erased, but all drawing on it is recreated according to the new coordinate system. If the mode was not already "world" when `setworldcoordinates` is called, there is a complete

reset of the window, and everything disappears. Either way the mode ends up set to "world".

This means that if, for example, you forgot about the unusual fact that positive y is up, and wrote an entire program that turns out to draw everything upside-down, you can add to the beginning of the program a call that turns things the other way up. Or for a tiny bit of entertainment, set the mode to "world" at the beginning of the program, then set the coordinate system the way your program was designed for at the end.

You can create your own turtle shapes by working out the coordinates of all of its corners (starting anywhere, but then moving around the shape in the proper order for drawing it), and put them in a tuple. Then register that tuple as a shape, and you can select it as normal with the `shape` method. The coordinates need to be arranged so that (0, 0) is where the turtle's centre should be when it is displayed, and they are measured in pixels always, regardless of any `setworldcoordinates`.

In this example, "pointy" is just a name I made up for the new turtle.

```
1 >>> shp = ((2.4, -20.0), (2.4, 0.0), (9.32, -4.0), (0.0, 12),
2 ...      (-9.32, -4), (-2.4, 0), (-2.4, -20), (2.4, -20))
3 >>> tu.addshape("pointy", shp)
4 >>> tu.shape("pointy")
```



The documentation says that the tuple should be of (x, y) pairs, but on my Windows PC that resulted in it pointing the wrong way. I had to make them (y, x) pairs instead, and that is what appears in the above example.

A slightly different approach is to define a polygon in exactly the same way as you prepare to fill a shape. First call `begin_poly()`, then move around the outline of the new shape, then call `end_poly()`. The way to register the shape after that is with:

```
tu.addshape("pointy", tu.get_poly())
```

again here, there is the problem of the (x, y) pairs getting switched around. On my Windows PC I have to move around what the edges of the shape would be if it were pointing downwards.

A third approach is just to make a .gif file that you would like to use as the turtle, register it, and select it like this:

```
tu.addshape("images\\E.gif")
tu.shape("images\\E.gif")
```

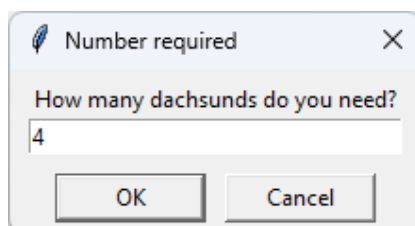
The centre of the gif will be at the turtle's actual position, and the coordinates do not get swapped around.

If input from the user is required, there are two turtle methods to help out. Both cause a pop-up dialogue window to appear, and do not return until the user enters something or cancels. What they return is what the user entered, or None if the user selected cancel or closed the dialogue window. Both take two strings as

their first parameters. The first is the label that will appear in the window's title bar, and the second is a prompt that appears as a question.

`textinput` takes no additional parameters, and returns the user's input as a string. `numinput` can take a few options and always returns a float. If what the user enters is invalid, it is rejected with a warning and the dialogue remains open. The options are `default = float`: this will appear in the input area as though the user had typed it. `minval = float` and `maxval = float`: these specify the range of acceptable replies, any value outside this range will be rejected in the same way as any other invalid input.

```
1 >>> import turtle as tu
2 >>> tu.numinput("Number required", "How many dachsunds do you need?",
3 ...           default = 1, minval = 0, maxval = 10)
4      4.0
```



Events such as keyboard keypresses and mouse clicks can be bound to callback functions that are automatically called when the events occur while the turtle window has focus. In the following, `fun0` represents a parameterless callback function, `fun2` is a two parameter function whose parameters (`x`, `y`) will be the screen coordinates for where the mouse was when the event occurred.

Beware: on my Windows PC, none of these things have any effect until I call the `listen` method. I only have to call it once and then it is permanently OK. This is not how it is supposed to be.

```
onkey(fun0, keyname)
onkeyrelease(fun0, keyname)
onkeypress(fun0, keyname)
```

`onkey` and `onkeyrelease` are two names for the same thing, their callback is called when the named keyboard key is released after being pressed. `onkeypress` triggers the callback when the key is first pressed. For most keys, `keyname` is just that key's character as a string, such as "x" or "X" or "%". It isn't just the character that appears on the key, it is the character that was typed, it can tell the difference between "x" and "X". "Return" and "\n" both work for the enter key, The others are "Up", "Down", "Left", "Right", "BackSpace", "Delete", "Escape", "Home", "Insert", "End", and "F1" to "F12". "Prior" and "Next" are for page up and page down. "Shift\_L", "Shift\_R", "Control\_L", "control-R", "Alt\_L", and "Alt\_R" are for the shift, control, and alt buttons on the left and right sides of the keyboard. On my Windows PC, "Shift\_R" doesn't do anything, but "Shift\_L" catches both shift keys. I can't find anything that works for the numeric keypad. There is no direct way to attach a callback to something

like control-x, but keeping track of the control keys being pressed and released lets you work out when control-x has been typed. If fun0 is None, the connection is removed.

```
listen(x = None, y = None)
```

Forces the focus onto the turtle window. The two parameters are never used, they exist only so that listen can be used as a fun2.

```
onclick(fun2, btn = 1, add = None)
```

```
onscreenclick(fun2, btn = 1, add = None)
```

```
onrelease(fun2, btn = 1, add = None)
```

onclick and onscreenclick are supposed to be two names for the same thing, but on my Windows PC, onclick has no effect at all. For onscreenclick the callback is called when a mouse button is pressed within the turtle window. For onrelease it is supposed to be called when the mouse button is released, but it also has no effect on my Windows PC. There is no onscreenrelease to make up for that. btn says which mouse button is to be listened for (1 means left, 2 the scroll wheel if there is one, and 3 right), and add is supposed to be True if you are adding an extra callback to the existing ones, or False or None if you are replacing the callback(s). Again, fun2 being None removes all callbacks for this button.

```
ondrag(fun2, btn = 1, add = None)
```

According to the documentation, this method is very badly named. The callback only has the standard (x, y) parameters, so it would not be capable of reporting a mouse drag. This method also has no effect on my Windows PC, so I can't find the truth by experiment.

```
ontimer(fun0, t)
```

Automatically calls fun0 after t milliseconds have elapsed.

According to the documentation the getcanvas() method should return a Canvas object as described in the next section, so you should be able to use it to set any of a Canvas' properties. But at least on my Windows PC, that doesn't work. Using Canvas methods does not cause an exception, but nothing else happens either.

## 39. Tkinter - the Canvas

Tk is a graphics system that has been around for quite a while, and has nothing to do with Python. tkinter is the Python interface to Tk. It comes with the standard Python installation. tkinter is very fond of keyword arguments so lines can get annoyingly long.

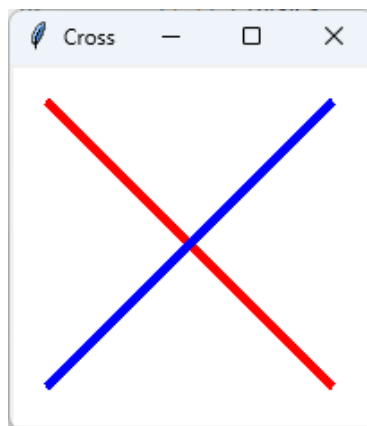
The documentation for tkinter is not very good. I have done what I can to search out as much as possible, but a few things remain a total mystery, and there is no guessing at what I might have missed altogether.

For displaying graphics, a `Canvas` is the most important kind of object, but for user interaction where you want to be able to enter text into a window, press buttons to make things happen, choose options from menus and so on, there are many other kinds of objects provided by `tkinter`. The objects, like `Canvas`s, that can be put in a `tkinter` window are supposed to be called `widgets`.

We'll start with the most basic of things, drawing a coloured diagonal cross. There are three steps:

1. Create a window and optionally give it a title.
2. Create a `Canvas` and put it in that window. A `Canvas` is a blank graphical component that can accept drawing commands.
3. Draw the two diagonal lines to make the cross.

```
1 >>> import tkinter as tk
2
3 >>> win = tk.Tk()
4 >>> win.title("Cross")
5
6 >>> ca = tk.Canvas(win,
7 ...             width = 200, height = 200,
8 ...             background = "white")
9 >>> ca.pack()
10
11 >>>
12 >>> ca.create_line(20, 20, 180, 180, width = 5, fill = "red")
    ca.create_line(20, 180, 180, 20, width = 5, fill = "blue")
```



An important note: Everything I do in this whole section is from `Idle`. If instead I run a Python program just by double clicking it I just see an empty window. Nothing happens unless I call `win.update()` after all the widgets have been created and put in place. `win.update()` does no harm when using `Idle`, so it is probably best to just use it always.

```
13
14 >>> win.update()
```

`Tk` is a class that represents windows. As soon as a `Tk` object is created, the blank window appears. As usual, it goes away when you click the close `x` button at the



top right. A window isn't the only thing that can contain graphical components, there are others. I'll use the term "container" to refer to anything of that kind. Measurements are in pixels unless you put a little effort into saying otherwise.

The parameters to `Canvas`'s constructor will be obvious apart from the first one. Whenever you create a graphical object (they call them widgets) apart from the top-level window itself, you must say which container it should go inside. That's the first parameter. But it is the `pack` method that actually puts it there. `pack` can be given parameters to say where you want it to be within its container. The default for `background` is system dependent. For me it is a vaguely ugly beige.

The call to `tk.Tk` on line 3 is not strictly necessary. As soon as you create any `tkinter` object, such as a `Canvas` as above, a window will automatically appear if you haven't already created one. The first `win` parameter to `tk.Canvas` can then be left out, the object will automatically belong to that window. While it saves a very short line of typing, it is usually not a good idea to take advantage of this. The object that `tk.Tk` creates can be used to control the window later on. For example you can say `win.destroy()` to make the window go away.

Once you have created the window with `tk.Tk()`, there are quite a few things you can do with it, you can resize it, minimise it, make it semi-transparent, give it your own icon, and a lot more. This is all covered in a later subsection called `The Tk window`.

The `create_line` method is probably obvious too, unless perhaps you are colour-blind. The first four parameters are `x0`, `y0`, the pixel coordinates for the beginning of the line, and `x1`, `y1`, the coordinates of the end. `create_line` returns a value that you can use later to refer to this particular line if you want to change its colour or something.

`create_line` can take any number of coordinate pairs, it connects each to the next with a straight line. It doesn't seem to make any attempt at antialiasing, so thin lines do not appear very prettily.

Keep in mind that this can all be done completely interactively. If a program needs to produce an image, then of course you'll have to code all the steps into it. But if you want to produce an accurate diagram, like the arrow shape a little below this, trial and error may be necessary. This is what I used:

```
1 >>> def start():
2 ...     global can, stk
3 ...     win = tk.Tk()
4 ...     can = tk.Canvas(win, width = 400, height = 400,
5 ...                     background = "white")
6 ...     can.pack()
7 ...     stk = []
8
9 >>>
10 ... def do(f, * p, ** k):
11 ...     global stk
12 ...     it = f(* p, ** k)
```

```

13         stk.append(it)
14 >>>
15 ... def kill(n = 1):
16 ...     global can, stk
17 ...     for i in range(n):
18 ...         it = stk.pop()
           can.delete(it)

```

The definition of `do` allows it to receive a function to be called, `f`, any number of positional parameters, `p`, and any number of keyword parameters, `k`. The global variable `can` is obvious, `stk` is a stack that records everything that has been drawn. The function is called, and the value it returns is saved on the stack. This value is a reference that can be used later to identify the thing that was drawn. So I can experimentally draw some lines or other things:

```

19 >>> start()
20 >>> do(can.create_line, 20, 20, 180, 180,
21 ...     width = 5, fill = "red")
22 >>> do(can.create_line, 180, 20, 20, 180,
23 ...     width = 5, fill = "blue")

```

and if I don't like the look of it:

```

24 >>> kill(2)

```

`can.delete` is the method that removes something from the Canvas, so `kill(n)` erases the last `n` things that were drawn. Once everything looks just right, I take a screen shot and I've got the image I needed.

There are a large number of predefined names for colours, far too many to list here, but the obvious web search always delivers. Actually, a few of them get it completely wrong. If you're told something is dark magenta but can see that it is a brownish green, don't just assume that `tkinter` has irrational colour names, just move on to the next search result. You can create any colour you like by specifying its red, green, and blue components as hexadecimal numbers between 0 and FF (0 to 255) after a # sign in a string. "#FF0000" would be the brightest red, "#FFFF00" is yellow, "#000066" is a dark blue, and so on. We are not limited to eight bit colour. 4, 12, and 16 bits are also allowed, but each of the R, G, B components must have the same number of bits. So if R, G, and B are hexadecimal digits, a colour can look like "#RGB", "#RRGGBB", "#RRRGGBBB", or "#RRRRGGGGBBBB".

An entire canvas can very easily be converted into a postscript (.ps) file. Some printers accept postscript files directly, but it is also very easy to convert a postscript file into a pdf file. You can also fairly easily convert postscript to any of the graphical file formats line .png, .jpg, .gif, etc. This will be covered quite soon under the heading Other Canvas methods as the `postscript` method.

## i. Lines

The first parameters to `create_line` are always an even number of numbers (floats are allowed) being the x, y coordinates of the start, corners, and end of the

line. They may be provided in many different ways. All of the following are permitted:

```
...(x0, y0, x1, y1, x2, y2, x3, y3, ...)
...((x0, y0), (x1, y1), (x2, y2), (x3, y3), ...)
...(((x0, y0), (x1, y1), (x2, y2), (x3, y3))), ...)
...([x0, y0], [x1, y1], [x2, y2], [x3, y3], ...)
...([[x0, y0], [x1, y1], [x2, y2], [x3, y3]]), ...)
...([(x0, y0), [x1, y1], (x2, y2), [x3, y3]]), ...)
...((([x0, y0], (x1, y1), [x2, y2], [x3, y3])), ...)
```

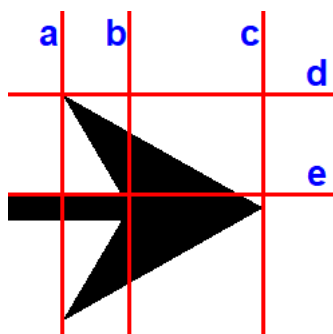
`create_line` can take a lot of options, all as keyword parameters. Apart from the already seen `width` and `fill` they are:

`arrow = "first" or "last" or "both"`

An arrow-head is drawn at the end of the line. "first" means that the arrow will be at the first point of the line, and "last" means it will be on the other end. "both" should be obvious.

`arrowshape = (len, back, wid)`

Specifies the exact shape and size of the arrow, which can be understood from this diagram:



The line labelled `e` shows where the end of the line would have been if there had been no arrow added to it. `len` is the distance between the `e` and `f` lines. `back` is the distance between the `d` and `g` lines. `wid` is the distance between the `a` and `b` lines. Note that `b` is on the edge of the drawn line, not at its centre.

The default values `(8, 10, 3)` work nicely for narrow lines, but they remain the same regardless of the width of the line, which doesn't sound like a very good plan to me. For wide lines you will always need to specify `arrowshape`.

`capstyle = "butt" or "projecting" or "round"`

Specifies the appearance of the endpoints of the whole potentially multi-point line. "butt" means that it will have normal square ends. "projecting" is also square, but extends the length of the line by half of its width. "round" means that the end will be a semicircle centred on the endpoint.

`joinstyle = "round" or "bevel" or "miter"`

Specifies the appearance of the places where two segments of a multi-point line meet at an angle. "round" means that it will be rounded off with a segment of a circle. "bevel" means that the corner will be cut off square. "miter" means that the corner will come to a sharp point, continuing the angles of the two line segments.

`smooth = True or False`

Only for multi-segment lines, the default is `False`. If `smooth` is `True`, instead of just drawing a sequence of straight lines between the specified points, it will make a nicely smooth curving line. The shape drawn is a "spline", which is a well-known thing in computer graphics, but far too complicated to go into here.

`splinsteps = n` only valid when `smooth` is also used

A spline is always drawn as a sequence of straight lines, and if those lines are short enough it looks smooth. The default is to always use 12 regardless of the length of the line segment. This lets you specify a different number.

`tags = "name"`

`tags = ("name1", "name2", "name3", ...)`

Tags allow related Canvas items to be controlled all at once. When *any* Canvas item is created, it can be given any number of tags. Later on, the Canvas' `itemconfigure` method can be used to change any of the options for all items with a particular tag in the same way. There is always a tag called "all" which means every item on the Canvas, you should not specify it as one of the names in this option.

`activefill = colour`

Many `tkinter` objects are aware of whether or not the mouse is on top of them. When it is, the object is considered to be active. It doesn't require any mouse clicking, but it does require exact placement over the object. If `activefill` is specified, then the line will turn to that colour when it is active, and back again when it isn't.

`activewidth = n`

Like `activefill` except that the line's width changes when it is active.

`activedash = a tuple of ints`

Like `activefill` but see the dash option above.

`disabledfill = colour`

The programmer may *disable* any object in a Canvas. This option, like `activefill`, causes the line to change colour when it is disabled.

`disabledwidth = n`

Like `disabledfill` except that the line's width changes when it is active.

`disableddash = a tuple of ints`

Like `disabledfill` but see the dash option above.

`state = "normal" or "hidden" or "disabled"`

"normal" is the default. If an item is created hidden, it is completely invisible. Disabled is the state that the previous three options react to. The state can be changed any time you want.

`dash = a tuple of ints`

The line will not be solid, but will have gaps along it, making a dashed line. The tuple can be any length you like. (12, 5, 9, 7) means that there will be an initial drawn section 12 pixels long, then a gap of 5 pixels, then a drawn section of 9 pixels, then a gap of 7 pixels, then a drawn section of 12 pixels again, then a gap of 5 pixels again, and so on, with the pattern repeating for the entire length of the line. If the length of the tuple is odd, the cycle will still blindly repeat it, so what represented a gap in one cycle will represent a solid section in the next. (10, ) means alternating lines and gaps all 10 pixels long.

This does not work under Windows. You will get a dashed line, but not the one you ask for.

`dashoffset = n` only valid when `dash` is also used

The dash pattern will start as though `n` (an int) pixels of the line have already been completed. With the (12, 5, 9, 7) example from above, a `dashoffset` of 10 would mean that the initial drawn segment is only 2 pixels long, but the pattern continues as normal after that.

Also does not work under Windows.

Recall that `create_line` returns a value, actually an int, that can be referred to later as a reference for the line. Given that number, a Canvas' `itemconfigure` method may be used to disable something, or make it disappear, or turn it back to normal, or a lot of other things. If `c` is a Canvas object and `n` is an item reference number then:

```
c.itemconfigure(n, state = "normal"), or
c.itemconfigure(n, state = "hidden"), or
c.itemconfigure(n, state = "disabled")
```

All of the options described above can be changed by the `itemconfigure` method. We've already seen `delete` removing things from a Canvas. Once something has been removed there is no getting it back.

```
c.delete(n)
```

Of course, lines are not the only things that can be painted onto a Canvas. There is also an `arc`, `bitmap`, `image`, `oval`, `polygon`, `rectangle`, `text`, and `window`. There is no `create_...able` item that represents an individual pixel, but a `PhotoImage` (described below) does allow you to set and read individual pixels within it.

## ii. Text

`create_text(x, y, other keyword options)` makes strings appear on the canvas close to position `(x, y)`. The options are:

`text = any string`

This is of course the string to be displayed. If it contains `\ns` they will behave as expected, making the text span multiple lines.

`anchor = "n", "s", "w", "e", "ne", "nw", "se", "sw", or "center"`

Gives meaning to the position coordinates `x, y`. "center" means that the text's bounding rectangle will be centred on `(x, y)`. "nw" means that `(x, y)` will be the top left corner of that rectangle, "s" means that `(x, y)` will be at the centre of its bottom, and so on. The default is "center".

`fill = colour`

Gives the colour of the text. No need to specify a background colour because only the text itself is drawn, the spaces around the letters are transparent. The default is system dependent.

`font = (family, size, style) or`

`font = family or`

`font = tkf.Font(options)`

In the first form, family is the font name, such as "times new roman" or "helvetica". Size is what you would expect, the font size in points. It can be in the form of an int or a string. A negative number is taken as the size in pixels instead of points. Style is optional. It is a string containing any combination of "italic", "bold", "underline", "overstrike" separated by spaces. "overstrike" means that a horizontal line is drawn across the text. If the font you ask for doesn't exist, one that seems as though it could be similar will be used instead.

In the second form, family is as before, just a font name as a string. This form is not recommended because you just don't know what size you are going to get.

The third form requires `import tkinter.font as tkf` before it will work. Using this style, you can create one Font object and use it for many strings. The options are all keyword parameters: family, size, weight ("bold" or "normal"), slant ("italic" or "roman" for unslanted), underline (0 or 1 for underlined), and overstrike (0 or 1 for overstruck).

The `tkinter.font` module provides some extra information. `tkf.families()` returns a tuple of all the font names or families on your system. If `f` is a Font object, `f.measure(string)` tells you how many pixels wide that string would be, and `f.metrics()` gives you a dictionary with four keys: "ascent", "descent", "linespace", and "fixed". The four values are all ints, `fixed = 1` means that it is a fixed width font, 0 means it isn't.

`justify = "left", "right", or "center"`

The default is "left". Only has an effect on multi-line strings. "left" means that all the lines will start at the same x position, "right" means they will end at the same x, "center" means they will all be equally balanced about the same x. "x" here does *not* refer to the x parameter.

`width = int`

The text is automatically split into multiple lines if it is wider than the given width.

Whenever you specify a size, except for a font size, you can specify the unit of measurement if you make it a string. "3.5i" means 3½ inches, "3.5c" means 3.5 centimetres, "3.5m" means millimetres, and "3.5p" means points (as in the traditional font measurement).

`tags, state, disabledfill, and activefill` mean the same as they do for lines.

### iii. Rectangles

`create_rectangle(x0, y0, x1, y1, other keyword options)` or  
`create_rectangle((x0, y0), (x1, y1), other keyword options)` or  
`create_rectangle(((x0, y0), (x1, y1)), other keyword options)` etc  
all draw a rectangle. `x0, y0` and `x1, y1` can be the coordinates of any two opposite corners, the order doesn't matter.

A rectangle must be perfectly aligned with the vertical and horizontal axes. If you need a solid rectangle at an angle, you can draw a thick line instead. If you just want an outline you're stuck, you'll have to work out the coordinates of the corners and draw four lines.

The options are:

`fill = colour` or

`activefill = colour` or

`disabledfill = colour`

The colour for the rectangle's enclosed area. The default is not to fill that area, having the effect of transparency.

`outline = colour` or

`activeoutline = colour` or

`disabledoutline = colour`

The colour for the outline of the rectangle. If colour is an empty string, no outline is drawn. The default is system dependent.

`width = size` or

`activewidth = size` or

`disabledwidth = size`

The width of the outline, default is 1.

tags, state, dash, and dashoffset are the same as for lines.

#### iv. Ovals and Circles

`create_oval(x0, y0, x1, y1, other keyword options)` or  
`create_oval([x0, y0], [x1, y1], other keyword options)` etc

The `x0`, `y0`, `x1`, `y1` are exactly as for rectangles and can be arranged in any of those three ways. A perfectly symmetrical oval is drawn, as big as it can be to fit inside the rectangle described by the coordinates. Because an oval is based on a rectangle, it must be perfectly aligned with the vertical and horizontal axes.

`create_arc` (just below) can do everything that `create_oval` can do and more, but `create_oval` is very slightly easier to use.

`fill`, `outline`, and `width` in their normal, active, and disabled forms are exactly the same as for rectangles.

tags, dash, dashoffset, and state are exactly the same as for everything.

#### v. Polygons

`create_polygon((x0, y0), (x1, y1), (x2, y2), ..., other keyword options)` or  
`create_polygon(x0, y0, x1, y1, x2, y2, ..., other keyword options)`  
or any of the options for presenting coordinates from `create_line`.

Like drawing a rectangle except that there can be any number of sides. The `xs` and `ys` are the coordinates of the vertices (corners). You do not need to repeat the first vertex at the end, the polygon will be closed automatically. Complex polygons, where the edges cross over each other, work as would be expected.

`fill`, `activefill`, `disabledfill`: polygons, unlike rectangles and ovals, are filled by default. If you only want the outline, set these colours to be empty strings.

`joinstyle` is the same as for lines: "round", "bevel", or "miter".

tags, dash, dashoffset, outline, activeoutline, disabledoutline, width, activewidth, disabledwidth, and state are the same as for everything else.

`smooth = True, False, "bezier", "raw", or ""`:

True means "bezier", False and "" mean no smoothing, just straight lines with corners. "bezier" is the same as `smooth = True` for lines. "raw" is like "bezier", but much more complicated. Too complicated for here. It is a cubic bezier curve with knot points and control points.

`splinsteps` is exactly the same as for lines.



## vi. Arcs

`create_arc(x0, y0, x1, y1, other keyword options)`

The `x0`, `y0`, `x1`, `y1` are exactly as for rectangles and can be arranged in any of those three ways. A perfectly symmetrical oval is imagined, as big as it can be to fit inside the rectangle described by the coordinates, and an arc or chord of that oval is drawn. Because an arc is based on a rectangle, it must be perfectly aligned with the vertical and horizontal axes.

`start = degrees`  
`extent = degrees`

The size of the arc or chord is determined by two angles. `start` is the direction from the centre to one of the edges, measured in degrees anti-clockwise from the positive x axis. That is right, or East, or three o'clock. Default zero. `extent` is how many degrees the arc stretches, anti-clockwise from the starting angle. It may be negative in which case it goes clockwise instead. The default is 90 degrees.

`style = "arc", "pieslice", or "chord".` Default is "pieslice".

The shape drawn always includes the portion of the outline of the oval described by `start` and `extent`, and with "arc" that is all that is drawn. With "chord" the two ends of the arc are also joined with a straight line. With "pieslice" there are two straight lines, one from the centre of the oval to one end of the arc, and one from the centre to the other end.

`fill`, `activefill`, `disabledfill` are the same as for polygons, the default is usually to fill the shape, but if `style` is "arc" there is nothing to fill, just a curved line.

`tags`, `dash`, `dashoffset`, `outline`, `activeoutline`, `disabledoutline`, `width`, `activewidth`, `disabledwidth`, and `state` are the same as for everything else.

## vii. Images from graphical files

An image represents the sort of thing that is stored in a graphics file, like a jpeg, a gif, or a png. An extra class, `PhotoImage`, is required to handle that side of things.

`create_image(x, y, other keyword options)`

The `x` and `y` are the same as for `create_text`, and are interpreted according to an anchor which is also the same. There are only two other options.

`tags` and `state` are exactly the same as for everything else.

`image`, `activeimage`, and `disabledimage` may be set to a `PhotoImage` object. There is a second alternative, a `BitmapImage`, but that is restricted to strictly two-colour images. Not even a grey scale, just one bit per pixel.

Once you have found your image file, turning it into a `PhotoImage` can be as simple as

```
1 >>> im = tk.PhotoImage(file = "pictures\\diagram.png")
    and displaying it on the canvas
2 >>> can.create_image(20, 20, anchor = "nw", image = im)
```

Some care is needed. If you create a `PhotoImage` and store it in a local variable in a function, then as soon as that function exits the `PhotoImage` becomes eligible for garbage collection. Being displayed in a `Canvas` isn't enough to keep it alive. Make sure the variable holding your `PhotoImage` lasts at least as long as the `Canvas` that displays it.

That may be the reason behind something that is bound to cause trouble. If I combine those two statements into one:

```
1 >>> can.create_image(20, 20,
2 ...     anchor = "nw",
3 ...     image = tk.PhotoImage(file = "pictures\\diagram.png"))
    absolutely nothing happens.
```

Unfortunately, a `tkinter PhotoImage` can only handle four formats: PNG, GIF, PGM, and PPM, and who has ever even heard of the last two? PGM is for grey-scale images, and PPM is an ancient thing from the very first days of the internet. There was some concern about GIF over a patent on an algorithm that it uses, and JPEG hadn't been invented yet. There is a third party module that handles JPG, BMP, and others. We'll come to that later.

Once a `PhotoImage` has been created, its size in pixels is given by `im.height()` and `im.width()`. You can inspect individual pixels, `im.get(x, y)` returns a three-tuple (R, G, B) of the colour components of the pixel, they will be in the 0 to 255 range.

When specifying a position within a `PhotoImage` with `column`, `row` or `x`, `y` coordinates, the coordinates are relative to the top left corner of the `PhotoImage`, not of the `Canvas`.

To change the colour of individual pixels:

```
1 >>> im.put("red", (30, 50))
    Sets the single pixel at x = 30, y = 50.
2 >>> im.put("blue", (30, 50, 70, 70))
    Sets the entire rectangle where x is between 30 and 70, and y is between 50 and
    70
3 >>> im.put("red red red green green", (30, 50, 70, 70))
    Fills the entire rectangle. The first three rows of pixels will be red, the next two
    rows will be green, and then the pattern will repeat until the rectangle is filled.
    The colours don't all have to be in one string, a list or tuple like ["red", "red",
    "red", "green", "green"] would do the same.
4 >>> a = ["red"] * 10 + ["green"] * 5 + ["blue"] * 15
5 >>> b = ["yellow"] * 20 + ["pink"] * 10
6 >>>
```

```
7 >>> patt = [a] * 10 + [b] * 8
      im.put(patt, to = (200, 200, 350, 350))
```

A two dimensional list or tuple gives a two dimensional pattern. `a` describes a row of 10 red pixels, then 5 green pixels, then 15 blue pixels, repeated as many times as necessary. `b` represents a similar row with 20 yellow and 10 pink pixels making the repeated pattern. The whole rectangle will be filled with the first 10 rows following pattern `a`, the next 8 rows following pattern `b`, and that repeating all the way to the bottom.

A useful feature is that you can make changes to a `PhotoImage` even after it has been put on a `Canvas`, and those changes will still appear immediately.

Individual pixels can be made transparent. `im.transparency_set(x, y, v)` makes the pixel transparent if `v` is `True`, opaque otherwise. Unlike with most graphics systems, there are no degrees of transparency, it's all or none. There is also no option for making a whole region transparent, you need a loop. You can check the transparency of a pixel with `im.transparency_get(x, y)`. Transparency just means that whatever is underneath the `PhotoImage` at that point gets seen instead. Usually that will be the `Canvas`' background colour. The entire image can be made transparent all at once with `im.blank()`.

The entire `PhotoImage`, with whatever changes you may have made to it, can be saved to a file:

```
1 >>> im.write("abc.png", format = "png")
```

The first parameter, the file name, can be anything you want. The format parameter must be one of "png", "gif", "pgm", or "ppm".

Finally, two more ways to create a `PhotoImage`. First you can create a totally blank one if you want to create an image from scratch, no need for a graphics file with e.g. `im = PhotoImage(height = 100, width = 100)`. There is no way to specify the background colour, but on my computer it comes up white.

Second, if you have read the entire contents of a supported graphics file into a bytes object, you can give that to the constructor.

```
1 >>> f = open("abc.png", "rb")
2 >>> cont = f.read()
3 >>> f.close()
4 >>> im = PhotoImage(data = cont)
```

The `data` parameter may also be a base 64 encoded bytes object as might be extracted from an email.

## viii. Other Canvas methods

Apart from the above methods for creating `Canvas` items, there are a large number of other methods that act on a whole `Canvas`. Keep in mind the fact that every

object is assigned an int value to identify it when it is created, and every `create...` method has a `tags` option that lets you attach tag strings to the created object. In the following descriptions, when a parameter is given as “items”, it can either be an int to refer to a single object or a tag to refer to any number of them. If a parameter is given as “singleitem” it can still be an int or a tag, but if that delivers more than one item, then only the item with the lowest int identifier is actually used.

There is also a thing called the display list. It prioritises all of the items in a Canvas. When any item is created, it goes to the top of the display list, anything below it is considered to be closer to the surface of the Canvas, and will therefore be obscured by the newer item if there is any overlap. The order of the display list can be changed, so this is not the same as an item’s int identifier.

`c.itemconfigure(items, option = value, option = value, ...)`

All of the identified items have their options changed or set as directed.

`c.itemconfigure(singleitem)`

Returns a dictionary where the keys are option names and the values are option values of all the options that the item has.

`c.itemcget(singleitem, option)`

Returns the value of the given option for this item. Option must be a string.

`c.move(items, dx, dy)`

`dx` and `dy` are distances, ints mean pixels but “0.5i”, “3c”, etc also allowed. All of the items are moved `dx` to the left, and `dy` down.

`c.delete(item)`

The item is removed. Not just made invisible, but permanently gone.

`c.type(singleitem)`

Returns what kind of item it is: “arc”, “image”, “line”, “oval”, “polygon”, “rectangle”, “text”, or “window”.

`c.postscript(option = value, option = value, ...)`

Converts the canvas into postscript (.ps) format. This can be used for printing or be converted to any of the commonly used graphics file formats.

`file = string`

The name of the file that the results should be stored in. You should make it end with `.ps` otherwise your computer might not know what to do with it. If you don’t provide a file name, the entire result is returned as a string.

`x, y, height, width`, all should be numbers

The default behaviour is that the output should only contain the part of the Canvas that is currently visible. If you have scrollbars, that is not going to be what you want. These four values should correspond

to the values you set for the `scrollregion` option (described under `canvas` options) when you created the Canvas: `x = left`, `y = top`, `height = bottom - top`, `width = right - left`.

`colormode = "color", "gray", or "mono", the default is "color"`

How to handle colours. "color" means take them as they are, "gray" means throw away colour information but keep the darkness and brightness producing a greyscale image, "mono" means just pure black and white, nothing in between.

`rotate = False`

False: produce the result in portrait style, True: landscape.

To make use of the `.ps` file, you will probably need to convert it to a different format. On this computer, double-clicking on the file automatically starts up Acrobat Distiller, which will convert it into a pdf. Usually the conversion happens immediately, but occasionally I have to select "open" from the menu and manually select the file.

If you don't want a pdf, there are numerous free on-line sites that will do the conversion to just about any graphics format file for you. I found that some of them somehow ignore the Canvas' background colour and render it as black. If that happens, just make sure the first thing you create on the Canvas is a rectangle that covers the whole thing with its `fill` set to the background colour you want.

There is a program called `ghostscript` which you can download for free (for non-commercial purposes) so that you don't have to use the on-line services. Unfortunately I have been unable to make it work, it always claims that some file doesn't exist, but won't say which one.

`c.scale(items, xOffset, yOffset, xScale, yScale)`

All of the items are enlarged or shrunk. Just changing their sizes isn't likely to be what you want, you'll almost certainly want the spaces between them to change in the same way. For that to work, you need to designate a centre point around which the scaling will happen. (`xOffset`, `yOffset`) provides that point. Every point in all of the items is moved so that its horizontal distance from the centre point is multiplied by `xScale`, and the vertical distance by `yScale`. The most common thing is that you want to scale the entire Canvas. In that case, select the top left corner as the centre point, and use "all" as the tag name.

`c.tag_bind(items, event, function = None, add = None)`

`c.tag_unbind(items, event, funcId = None)`

These will be saved until the subsection on binding events.

`c.scan_mark(x, y)`

`c.scan_dragto(x, y, gain = 10)`

These will be saved until the subsection on scrolling.

`c.bbox(items = None)`

Bounding Box. Finds the smallest rectangle that encloses all of the items, and returns its edges as a four-tuple of pixel numbers (left, top, right, bottom).

If the parameter is None, then every item on the Canvas is included.

`c.coords(singleitem)`

Returns a list of the coordinates associated with the item. The list is an alternating sequence of x1, y1, x2, y2, ... If the item is a line or a polygon, these are the coordinates of its start, corners, and end. For anything else they are just four numbers giving its bounding box.

`c.canvasx(x, gridspacing = None)`

`c.canvasy(y, gridspacing = None)`

x or y are positions relative to the top left corner of the window. what is returned is the equivalent position relative to the top left corner of the canvas. If gridspacing is not None, the result is rounded to be a multiple of it.

`c.gettags(items)`

If items only refers to a single item, return a tuple of all the tags that item has. Otherwise, if items is a tag, it does the same for the item that has that tag and is lower in the display order than the others.

`c.dtag(items, unwantedtag)`

The unwanted tag (a string of course) is taken away from all of the items.

`c.find_all()`

Returns a list of the int identifiers for all of the items on the Canvas.

`c.find_withtag(items)`

Returns a list of the int identifiers for all of the items.

`c.find_enclosed(xleft, ytop, xright, ybottom)`

Returns a tuple of the int identifiers for every item that is entirely inside the given rectangle.

`c.find_overlapping(xleft, ytop, xright, ybottom)`

Returns a tuple of the int identifiers for every item that is either entirely inside the given rectangle, or crosses its edge.

`c.find_closest(x, y, halo = None, start = None)`

Normally returns the int identifier for the object that is closest to the given x, y position. The result is a one-tuple normally, but a zero-tuple if there is no qualifying item. The halo parameter doesn't seem to have any effect. The start parameter is supposed to have some effect, but I haven't been able to detect it. Perhaps it's another Windows thing.

`c.find_above(singleitem)`

`c.find_below(singleitem)`

Finds the item that is closest to but just above/below the given item in the display list as a tuple of one int identifier, or a zero-tuple if there is nothing above/below. The documentation says something about what happens if multiple items qualify, but if we're talking about the closest in the display list, that can't really happen.

`c.tag_raise(items, singleitem)`

`c.tag_lower(items, singleitem)`

Changes the positions within the display list of all of the items, so that they are immediately above/below the given single item. If there are multiple items, their relative ordering is kept. This means that if one of the items was obscuring or obscured by some other item, the display may change.

The `c.addtag_...` methods all add a new tag to a group of existing items. In the following, `newtag` is the name of the tag to be added.

`c.addtag_all(newtag)`

Add the new tag to every item in the whole Canvas.

`c.addtag_enclosed(newtag, x0, y0, x1, y1)`

Add the new tag to every item that is completely inside the rectangle with top left corner `(x0, y0)` and bottom right corner `(x1, y1)`.

`c.addtag_enclosed(newtag, x0, y0, x1, y1)`

Add the new tag to every item that is either completely inside the rectangle with top left corner `(x0, y0)` and bottom right corner `(x1, y1)` or crosses its edge.

`c.addtag_closest(newtag, x, y, halo = None, start = None)`

Add the new tag the item that is closest to position `(x, y)`. As with `find_closest`, `halo` and `start` are a mystery.

`c.addtag_withtag(newtag, items)`

Add the new tag to all of the items.

`c.addtag_above(newtag, singleitem)`

`c.addtag_below(newtag, singleitem)`

Add the new tag to the one item that is immediately above or below the given single item in the display list.

## ix. Canvas options

When creating a Canvas there are more available options than the `width`, `height`, and `background` that we've seen. You do not need to provide all the options to the constructor. Once a `tkinter` object exists you can add or change any of the options with the `configure` method (below).

`background = colour`            `background` may be abbreviated to `bg`

`borderwidth = size`            `borderwidth` may be abbreviated to `border` or `bd`  
The Canvas will be enlarged by the given size in all four directions to create a border. Drawing is not allowed in the border, but coordinates are offset so that nobody has to do anything about it, (0, 0) is the top left drawable position. The border remains clear, in the Canvas' background colour. Under windows, it doesn't work quite properly, the borders are not even.

`relief = "groove", or "raised", or "ridge", or "sunken"`.  
Only meaningful in conjunction with `border`, and only effective for narrow borders of about 4 pixels or so. Attempts to give a 3-D look to the border.

`closeenough = distance`  
Normally a component only takes on its active colour etc when the mouse is directly above it. This changes that. A component becomes active if the mouse is within that many pixels of it.

`scrollregion = (left, top, right, bottom)`  
`xscrollincrement = distance`  
`yscrollincrement = distance`  
These are all part of setting up scrollbars for your Canvas, and are described later in the subsection on scrolling.

`cursor = string`  
If this is set, then the shape of the mouse cursor changes whenever the mouse is within the bounds of the canvas. On my windows computer, most of the possibilities are very badly drawn indeed. I'm led to believe that this is mostly an Xwindows feature and things might be better under that. This is my categorisation of the available cursors as I see them on this system:

These cursors are well drawn, and are likely to be useful:

- "arrow": the normal arrow pointer used to select things,
- "crosshair": a sort of plus sign for more accurate positioning,
- "fleur": arrows pointing in all four direction,
- "hand2": a pointing finger,
- "question\_arrow": the same as "arrow" but with a question mark,
- "tcross": the same as "crosshair" but hollow,
- "watch": an hourglass, used to indicate "wait",
- "xterm": the vertical text cursor.

These are at least reasonably well drawn, but not very useful:

- "bogosity", "bottom\_right\_corner", "center\_ptr", "cross", "plus",
- "sb\_h\_double\_arrow", "sb\_v\_double\_arrow".

These are of at best moderate quality, but could be useful:

- "based\_arrow\_down", "based\_arrow\_up", "bottom\_left\_corner",
- "bottom\_side", "bottom\_tee", "diamond\_cross", "dotbox",
- "left\_tee", "ll\_angle", "lr\_angle", "right\_tee", "rtl\_logo",
- "sb\_down\_arrow", "sb\_right\_arrow", "sizing", "top\_left\_corner",



```
"top_right_corner", "top_side", "top_tee", "ul_angle", "ur_angle",  
"X_cursor".
```

These are just plain bad:

```
"boat", "box_spiral", "circle", "clock", "coffee_mug",  
"cross_reverse", "dot", "double_arrow", "draft_large",  
"draft_small", "draped_box", "exchange", "gobbler", "gumby",  
"hand1", "heart", "icon", "iron_cross", "left_ptr", "left_side",  
"leftbutton", "man", "middlebutton", "mouse", "pencil", "pirate",  
"right_ptr", "right_side", "rightbutton", "sailboat",  
"sb_left_arrow", "sb_up_arrow", "shuttle", "spider", "spraycan",  
"star", "target", "top_left_arrow", "trek", "umbrella".
```

## 40. Universal widget methods

All tkinter gui objects, or widgets, also have a large number of methods. The following are available with every kind of widget, and are only a selection of the most useful ones, there are too many.

`w.configure(keyword = value)`

Changes an option, e.g. `w.configure(background = "yellow")`. Note that the option names are not strings this time, they are keyword parameters, just as they were for the widget's constructor.

`w.cget(string)`

Returns the value of a particular option, e.g. `w.cget("background")` might return "yellow". This will nearly always be the same string or int that you originally supplied, but for more complex values like fonts it will not. `cget` only delivers option values that you set: every widget has a width, but `w.cget("width")` will return 0 unless you explicitly set the width option.

`w.keys()`

Returns a list of strings which are the names of all the options that are applicable to this kind of widget.

`w.winfo_rgb(string)`

The string must be the name of a colour. Returns a three-tuple of that colour's red, green, blue values on an unsigned 16 bit scale (0 to 65535), for example `w.winfo_rgb("yellow")` is (65535, 65535, 0). What isn't this a class method?

`w.winfo_geometry()`

Returns a string the widget's size and position. It will always be of this format '71x105+41+253' which means width = 71 pixels, height = 105, left edge is 41 pixels from its containing window's left edge, and top is 253 pixels down from its window's top. Unlike `w.cget("width")`, this returns the correct current values even if you didn't set them.

`w.winfo_width()`, `w.winfo_height()`, `w.winfo_x()`, and `w.winfo_y()`

Return the same values as `w.wininfo_geometry()` but as more conveniently usable ints.

`w.after(milliseconds, callback, parameters for callback)`

After at least `milliseconds` has expired, the `callback` function will be called with the provided parameters. They should be just separate parameters, not a tuple unless your function wants a tuple. `after` returns a string which may be used to cancel this request. `Callback` is optional. If only `milliseconds` is given, this method will just wait for the given time before returning.

`w.after_cancel(string)`

Cancel the `after` request that returned this string.

`w.after_idle(callback, parameters for callback)`

Call the given function the next time the tkinter system finds itself idle, having no events to deal with. There are never any events to deal with unless you have started an event loop.

`w.focus_get()`

Focus refers to the widget that is currently selected for input operations. In many gui systems, when a button has focus, pressing enter or space has the same effect as clicking on it. If a text input box has focus, everything you type will go into that box. `focus_get` returns the widget object that currently has focus, or `None` if none have.

`w.focus_set()`

If any of the current program's widgets has focus, focus will immediately be transferred to `w`. If another program currently has focus, then focus will be set to `w` as soon as this program regains control again.

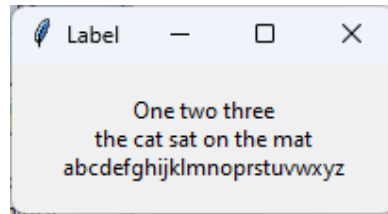
`w.focus_force()`

Is the same as `focus_set`, except that it is more demanding. Focus will be set on `w` even if it is currently held by another program.

## 41. Label

A Label is the most basic kind of thing that can appear in a window. It is just a piece of text.

```
1 >>> win = tk.Tk()
2 >>> win.title("Label")
3 >>> lab = tk.Label(win,
4 ...             text = "One two three four\nthe cat sa"
5 ...             "t on the mat\nabcdefghijklmnop"
6 ...             "rstuvwxyz", padx = 25, pady = 15)
7 >>> lab.pack()
```



Note that I only provided four parameters to `Label`. Those are not three strings after `text =`, I am making use of a Python feature: if multiple strings appear one after the other with only white space between them, they are treated as a single string. Without that I wouldn't be able to do this demonstration satisfactorily. And don't forget about the universal methods: `configure` and so on.

The `padx` and `pady` parameters say that the size of the `Label` object will not just be the minimum size required to hold the text, but an extra 25 pixels will be added to the left and right, and 15 above and below.

Other options are

`background = colour`                      `background` may be abbreviated to `bg`  
`foreground = colour`                      `foreground` may be abbreviated to `fg`

Specify the background and foreground colours. `fg` is the colour of the text, and `bg` is the colour of the space surrounding it. Colours are described exactly as they are for `Canvas` components.

`width = size`  
`height = size`

Allows you to specify an exact size, over-riding the content-based value. But beware, the units are not obvious. If a component contains text, as `Labels` usually do, the size is not measured in pixels but in characters. For height that's fine, but unless you are using a fixed width font, width is impossible to judge. If the component contains an image then the height and width are taken to be in pixels.

`anchor = "n", "s", "w", "e", "ne", "nw", "se", "sw", or "center"`

If you force a label to be bigger than it naturally would be, perhaps by specifying `height` and `width`, the `anchor` says where the text will appear. The default is `"center"`. `"nw"` means it will be snug up against the top left corner of the `Label`, and so on.

`image = PhotoImage`

`PhotoImage` should be a `PhotoImage` object as discussed for `Canvas`. Any `text` parameter will be ignored unless you set `compound`. The `Label` will simply show the image in place of the normal text.

`compound = "bottom", "top", "left", "right", or "center"`

Any of these settings means that the `Label` can have both text and an image. The first four give the `Image`'s position relative to the text, so `"bottom"`

means the text will appear like a title, above the image. "center" means that the text will be superimposed on the image at its centre.

`state = "normal", "disabled", or "active"`

The state of a Label is supposed to automatically change to "active" whenever the mouse cursor hovers above it, regardless of any button pressing. On my Windows computer that doesn't work. "disabled" is not very useful for Labels because it is meant to make a component unresponsive to mouse clicks, and Labels already are. Background and foreground colours may change according to the state of the object. "normal" is the default.

`activebackground = colour`

`activeforeground = colour`

`disabledforeground = colour`

Specify the alternate colours used when the object's state is not "normal". Why on earth is there no `disabledbackground`?

`wraplength = number`

Specifies a maximum length for a line of text. If a line is longer than this a newline will be inserted, preferably replacing a space, but anywhere if necessary. The documentation says that we specify the number of characters, but experiment shows that it is really the size, in pixels by default.

`underline = int`

The specified character of the string, counting from zero, will be underlined. This is usually used to remind users of "hot keys", keyboard shortcuts that have the effect of clicking on a button. Therefore it is not much use with a Label. The default, -1, means no underlining.

`cursor, borderwidth, border, bd, and relief`

Exactly as for Canvas.

`justify = "left" or "right" or "center"`

`font = family`

`font = (family, size, style)`

`font = tkf.Font(options)`

Exactly as for `Canvas.create_text`.

## Automatic line breaking

A Message is almost identical to a Label. The only difference is that if you want a Label to have more than one line of text, you must explicitly put `\n` characters in its text. A Message automatically breaks lines to fit them into the desired width.

There is a slight difference in the options. There is no height, and width is measured in pixels rather than a Label's number of characters. With a Message,

if you specify a width then the height is completely controlled by the contents, it can't take a `Scrollbar`. Alternatively, you can specify the `aspect` option as a number. This controls the shape of the widget. 100 means exactly square, 200 means twice as wide as it is high, 50 means twice as high as it is wide, and so on.

## Automatic updating

`tkinter` defines a `StringVar` class. The objects hold strings, and when passed to a function give the effect of call by reference. The function can change the value of the `StringVar`'s string so that the new value can be seen when the function exits.

`StringVars` may only be created when a `tkinter` window is in existence. For some reason they have to be associated with a window. The constructor is:

```
s = StringVar(master = None, value = None, name = None)
```

`master` is the window that it is associated with. `None` means the currently active one. `value` is a string giving its initial value, and `name` allows you to name the variable so you can tell the difference between a number of `StringVars`. The default is to make up a unique name.

`StringVars` have the obvious `set` and `get` methods, `s.set("new value")` changes the value, `s.get()` returns the current value.

`StringVars` have built-in assistance for debugging. You can specify a callback function that will automatically be called whenever a `StringVar`'s value is changed, deleted, or even looked at. Deletion happens when you say `del s`. The callback function must have three parameters. The first will be the name of the `StringVar`, the second isn't of much use, and the third is the operation that caused the callback, "r", "w", or "u".

```
1 >>> def callbk(n, x, o):
2 ...     print(o, "operation on", n)
3 >>> sv = tk.StringVar()
4 >>> sv._name
5     'PY_VAR31'
6 >>> sv.set("hello")
7 >>> sv.get()
8     'hello'
9 >>> sv.trace("w", callbk)
10     '1605411050432callbk'
11 >>> sv.trace("u", callbk)
12     '1605345043281callbk'
13 >>> sv.get()
14     'hello'
15 >>> sv.set("goodbye")
16     w operation on PY_VAR31
17 >>> sv.trace_remove("write", "1605411050432callbk")
18 >>> sv.set("again")
19 >>> del sv
20     u operation on PY_VAR31
```

The `trace` method's first parameter is "w" if you want to detect `.set` operations, "r" if you want to detect `.get` operations, and "u" if you want to detect deletions. `trace` returns a reference string which you must remember if you ever want to cancel the trace. That's what `trace_remove` does. Mysteriously its first parameter must be the entire word "read", "write", or "unset", it doesn't accept the abbreviations "r", "w", or "u" given to `trace`.

There are also `IntVar`, `BooleanVar`, and `DoubleVar` classes, but the option is still called `textvariable` for all three.

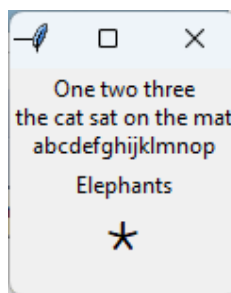
To make use of a `StringVar` in a `Label`, forget about the `text = "string"` parameter, and use the `textvariable = sv` parameter instead. The `Label` will show the value of `sv` as its text, and it will be updated automatically whenever `sv`'s value changes.

```
1 >>> sv = tk.StringVar(value = "Elephants")
2 >>> lab = tk.Label(win, textvariable = sv)
```

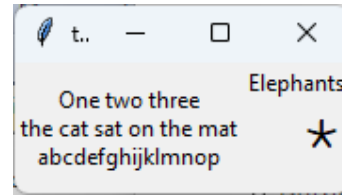
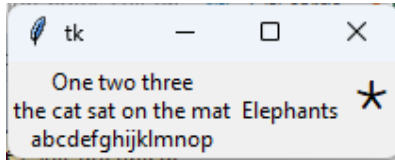
## 42. Multiple items in a window

### i. Pack

```
1 >>> win = tk.Tk()
2 >>> lab1 = tk.Label(win,
3 ...           text = "One two three four\nthe cat sa"
4 ...           "t on the mat\nabcdefghijklmnop")
5 >>> lab1.pack()
6 >>> lab2 = tk.Label(win, text = "Elephants")
7 >>> lab2.pack()
8 >>> lab3 = tk.Label(win, text = "*",
9 ...           font = ("courier new", 30))
10 >>> lab3.pack()
```



You can `pack()` any number of things, but they all end up on top of each other. We can improve in this just a little bit by using `pack`'s optional `side` parameter. The possible values are "top" (the default), "bottom", "left", and "right". It says which edge of the window the object wants to be next to. If more than one object requests the same side, they are lined up, moving inwards from the desired side. The side only specifies one dimension, the object is centred in the other dimension.



The first example shows what happens when all three labels request "left", the second is when the first chooses "left", the second chooses "top", and the third chooses "right". It isn't easy to predict exactly what will happen.

Other keyword options for `pack` are

```
padx = number
pady = number
ipadx = number
ipady = number
```

All four of these add extra space around the text. The `x` versions add the given amount both to the left and to the right, and the `y` versions both above and below. If you don't do anything with background colours, you might not see any difference between the plain versions and the `i...` versions. The plain versions put space around the outside of the entire `Label` object, so the window's background colour will show through in this space. The `i...` versions (`i` is for internal) make the `Label` object itself larger, so the extra space is filled with the `Label`'s background colour. You can use all four together.

```
expand = True or False           False is the default
fill = "none", "x", "y", or "both"  "none" is the default
```

At least on my PC, these two only work if used together, so that is what I'll describe here. Normally, if you resize the window to make it larger, objects within it will move around to remain stuck to their desired sides, but will not grow. `expand` and `fill` change that. If `fill` is "x" the object, along with its background colour, will stretch horizontally as far as it can. "y" lets it stretch vertically, and naturally "both" lets it grow both ways. Growth also happens as soon as the window is fully populated by its packed objects.

This is definitely not what the documentation says will happen, so possibly it will be different on other types of system.

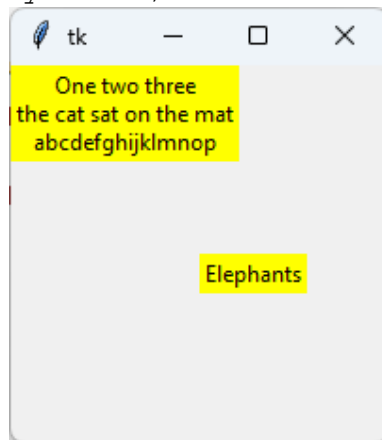
```
anchor = "n", "ne", "e", "se", "s", "sw", "w", "nw", or "center", the default
```

It is very hard to tell what this is supposed to do. `side` continues to determine which side of the window components will grow from, but `anchor` seems to change the side that they will stick to if the window grows. Maybe that is indeed what it is supposed to do.

## ii. Place

Fortunately, `pack()` isn't the only way to put things into a window. Another method, `place()` allows more control but is very inflexible, you have to have a good idea of how big all of your objects are going to be, and a slight redesign can require a lot of changes. A third alternative, `grid()` is the easiest to design with, but I'll save that for last.

```
1 >>> win = tk.Tk()
2 >>> lab1 = tk.Label(win, text = "One two three four\nthe cat s"
3 ...           "at on the mat\nabcdefghijklmnop"
4 ...           "nop", bg = "yellow")
5 >>> lab1.place(x = 0, y = 0)
6 >>> lab2 = tk.Label(win, text = "Elephants", bg = "yellow")
7 >>> lab2.place(x = 100, y = 100)
8 >>> lab3 = tk.Label(win, text = "*",
9 ...           font = ("courier new", 30),
10 ...          bg = "yellow")
11 >>> lab3.place(x = 300, y = 200)
```



The `x` and `y` parameters to `place` tell it the coordinates for the top left, or "nw", corner of the object. Notice that only the first two labels are visible. When `pack` is used, the window grows to fit its contents. When `place` is used, it doesn't. If I stretch the window, the missing Label appears. That is easily fixed:

```
1 >>> win.geometry("500x400")
```

That sets the window to 500 pixels wide and 400 pixels high. Curiously, `geometry` only accepts strings, you can't give it the two numbers. The string can be extended to this form "`widthxheight+left+top`". `width` and `height` are as before, `left` and `top` give the position of the window on the screen. Calling `geometry` with no parameters returns the full string giving the window's current size and position.

And while we're at it, `win.destroy()` will make the window go away, you don't have to wait for the user to close it manually.

Naturally, `place` has other options:



`anchor = "n", "ne", "e", "se", "s", "sw", "w", "nw", or "centre"`  
"nw" is the default.

This says how to interpret the `x` and `y` parameters. If `anchor` is "sw" then it will be placed so that its South-West (bottom left) corner is at position (`x`, `y`).

`width = number`  
`height = number`

Let you over-ride the object's natural size.

`relx = float between 0 and 1`  
`rely = float between 0 and 1`

An alternative to `x` and `y`. `relx = 0` refers to the left edge of the window, `relx = 1` is the right edge, `relx = 0.5` is the exact horizontal centre of the window, and so on. If you change the size of the window, the component will keep its relative position.

`relwidth = float between 0 and 1`  
`relheight = float between 0 and 1`

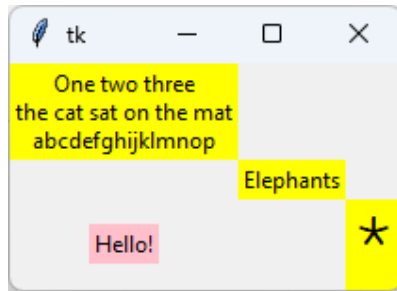
An alternative to `width` and `height`. `relwidth = 0.5` means that the width of the object will be half the width of the window.

`bordermode = "inside" or "outside"`

It is possible for windows and other containers like them to have borders around them. `bordermode = "inside"` means that position (0, 0) is just inside the border at the top left corner. `bordermode = "outside"` means that (0, 0) is the window's own real top left corner.

### iii. Grid

```
1 >>> win = tk.Tk()
2 >>> lab1 = tk.Label(win, text = "One two three four\nthe cat s"
3 ...           "at on the mat\nabcdefghijklm"
4 ...           "nopqrstuvwxyz", bg = "yellow")
5 >>> lab1.grid(row = 0, column = 0)
6 >>> lab2 = tk.Label(win, text = "Elephants", bg = "yellow")
7 >>> lab2.grid(row = 1, column = 1)
8 >>> lab3 = tk.Label(win, text = "*",
9 ...           font = ("courier new", 30),
10 ...          bg = "yellow")
11 >>> lab3.grid(row = 2, column = 2)
12 >>> lab3 = tk.Label(win, text = "Hello!", bg = "pink")
13 >>> lab3.grid(row = 2, column = 0)
```



The grid placement scheme imagines that the window is divided into some number of rows and columns, just like a spreadsheet. Each row and column is sized to fit the largest object it contains, and the window is sized to fit everything. The number of columns and rows is completely determined by the values you give for `row` and `column` when you call `grid`. They don't get specified in advance. Unless you specify otherwise, stretching or shrinking the window has no effect on the positions of its contents.

```
padx = number
pady = number
ipadx = number
ipady = number
```

These four options are exactly as they were for `pack()`.

```
sticky = string consisting of any combination of n, s, w, and e.
```

If the grid cell ends up being larger than this object, the object will be stuck to the given sides of the cell. If two of the specified sides are opposites, the object will be stretched so that it can stick to both. So "nw" means it will just sit in the top right corner at its natural size. "nse" means it will sit up against the right edge at its normal width, but its height will be stretched to the entire height of the cell.

```
rowspan = int, the default is 1
```

```
columnspan = int, the default is 1
```

Neighbouring cells will be merged into a single big cell capable of holding one larger than usual object. Two examples: If `row` is 1, `column` is 3, `rowspan` is 1 and `columnspan` is 5 then the five cells (1, 3), (1, 4), (1, 5), (1, 6), and (1, 7) become a single big cell. If `row` is 2, `column` is 3, `rowspan` is 2 and `columnspan` is 2 then the four cells (2, 3), (2, 4), (3, 3), and (3, 4) become a single big cell.

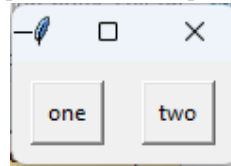
## 43. Button

The `Button` is the most basic of the interactive components. It has all the usual features for getting its appearance right, but it also needs something that hasn't come up before. We need some way of saying what should happen when the button is pressed. This is done by providing a "callback" as the `command` option. A callback is a parameterless function that is automatically called when some event, in this case a mouse press in the right place, occurs.

```

1 >>> def callback1():
2 ...     print("You pressed button one")
3 >>> def callback2():
4 ...     print("You pressed button two")
5
6 >>> win = tk.Tk()
7 >>> b = tk.Button(win, text = "one", command = callback1,
8 ...               activebackground = "yellow",
9 ...               padx = 5, pady = 5)
10 >>> b.grid(row = 0, column = 0, padx = 10, pady = 10)
11 >>> b = tk.Button(win, text = "two", command = callback2,
12 ...               activebackground = "cyan",
13 ...               padx = 5, pady = 5)
14 ...     b.grid(row = 0, column = 1, padx = 10, pady = 10)

```



It is immediately up and running, reacting properly to button clicks. It is beginning to look as though substantial programs will need a lot of callback functions, but if you remember `functools.partial` and how to define a class, there isn't necessarily such a crowd.

```

1 >>> import functools as ft
2
3 >>> class variable:
4 ...     def __init__(self, value):
5 ...         self.value = value
6
7 >>> def callback(butnum, var):
8 ...     print("you pressed", butnum, end = ", ")
9 ...     var.value += butnum
10 ...     print("the value is now", var.value)
11
12 >>> x = variable(0)
13 ...
14 >>> win = tk.Tk()
15 >>> b = tk.Button(win, text = "one",
16 ...               command = ft.partial(callback, 1, x),
17 ...               activebackground = "yellow",
18 ...               padx = 5, pady = 5)
19 >>> b.grid(row = 0, column = 0, padx = 10, pady = 10)
20 >>> b = tk.Button(win, text = "two",
21 ...               command = ft.partial(callback, 2, x),
22 ...               activebackground = "yellow",
23 ...               padx = 5, pady = 5)
24 >>> b.grid(row = 0, column = 1, padx = 10, pady = 10)
25 >>> b = tk.Button(win, text = "three",
26 ...               command = ft.partial(callback, 3, x),
27 ...               activebackground = "yellow",
28 ...               padx = 5, pady = 5)
29 >>> b.grid(row = 0, column = 2, padx = 10, pady = 10)
30

```

```
31     you pressed 1, the value is now 1
32     you pressed 3, the value is now 4
33     you pressed 2, the value is now 6
34     you pressed 1, the value is now 7
```

The options for a Button are

text, textvariable, justify, image, and compound  
Exactly as they are for Labels.

command = zero parameter callback function

background, bg, foreground, and fg  
Exactly as they are for Labels.

activebackground = colour  
activeforeground = colour

Almost the same as for Labels, but not quite. The active colour appears while the button is being clicked (i.e. between the down and up events), merely hovering the mouse has no effect. And they both work under Windows.

padx, pady, height, width, cursor, state, wraplength, underline, and anchor  
Exactly as they are for Labels.

borderwidth = number      borderwidth may be abbreviated to border or bd  
relief = string  
overrelief = string

border and relief are the same as for a Canvas. overrelief is allowed the same values as relief, the border shading will change to overrelief whenever the mouse is within the Button whether clicked or not.

repeatdelay = milliseconds  
repeatinterval = milliseconds

Unlike keyboard buttons, gui buttons do not normally repeat when they are held down. If you want a Button that does repeatedly call its command callback when held down, set these two options. repeatdelay is how long the Button needs to be held down before the first call happens. repeatinterval is how long to wait for the next call as long as the Button is held down.

Every Button also has two new methods, but don't forget about the universal methods: configure and so on.

b.flash()

The Button will flash, rapidly switching between its normal and active colour schemes three or four times.

b.invoke()

Simulates a click. The Button's callback function is called.

## 44. Entry - simple text input

An Entry is a box in which the user can type a single line of text that the program can then access. This is an ugly and slightly dangerous calculator:

```
1 >>> def doit():
2 ...     svout.set(eval(svin.get()))
3
4 >>> win = tk.Tk()
5 >>> svin = tk.StringVar()
6 >>> svout = tk.StringVar()
7 >>> inw = tk.Entry(win, width = 20, textvariable = svin,
8 ...                 background = "white")
9 >>> inw.pack()
10 >>> doitw = tk.Button(win, text = "do it now", command = doit)
11 >>> doitw.pack()
12 >>> outw = tk.Label(win, textvariable = svout,
13 ...                 background = "white")
14 >>> outw.pack()
```

The user types an expression into the Entry box at the top, clicks the Button in the middle, and the value appears in the Label at the bottom.

background, bg, foreground, fg, textvariable, justify, font, relief, borderwidth, border, bd, and cursor,

Exactly as they are for Labels.

width = size

Specifies the width of the box, measured in characters, not pixels. This will be inaccurate unless you use a fixed-width font. The default is 20.

state = "normal", "disabled", "readonly", or "active"

When the Entry is "disabled", the user can't do anything with it. "readonly" is similar, but the user can still select and copy the text. The programmer should not normally set the state to "active", it is supposed to change to "active" automatically when the mouse is inside the Entry, but it doesn't on my Windows PC.

disabledbackground = colour

disabledforeground = colour

Alternative colours used when the Entry object is disabled.

insertbackground = colour

insertwidth = number of pixels

insertontime = milliseconds

insertofftime = milliseconds

These change the appearance of the cursor that marks the position where keypresses will be entered. insertbackground is the cursor's colour,

default "black". `insertwidth` is the width of the cursor, default 2 pixels. The cursor normally flashes on and off fairly evenly, the two time options change the rate of flashing. `insertontime` is the time it will spend visible, and `insertofftime` is the time it will then spend invisible. An `insertofftime` of 0 means no flashing at all.

There is no `padx`, `pady`, `height`, `activebackground`, or `activeforeground`

`validatecommand` = tuple

`validate` = "focus", "focusin", "focusout", "key", "all", or "none"

The contents of the `Entry` can be automatically checked for validity as it is being changed. You provide a function that returns `True` if the contents is valid, and `False` if it isn't. You then register that function with your window, and provide it as the first item in `validatecommand`'s tuple. `validate` is set to say exactly when the automatic checking should be performed.

The values for `validate` mean:

"focus"	getting or losing focus
"focusin"	getting focus
"focusout"	losing focus
"key"	contents changed by a keyboard keystroke
"all"	any of the above
"none"	never validate

The easy way to do this doesn't work (on Windows anyway). The easy way would be to make your validating function just look at the `StringVar`'s value and base its decision on that. It doesn't work because the validator's job is to say whether the change should be allowed or not. If it says `False`, the `StringVar` doesn't get changed. That means that the validator is called before the `StringVar` changes, so it will be making its judgement based on what the string was *before* the user's change.

Instead you specify extra parameters that will be generated and supplied to your validator as parameters. That's what the other entries in the tuple are. I have only found one of them to be really useful, but I can imagine a use for three others.

"%P" is what the string would be if the change is allowed.

"%s" is the unchanged string, the same as the `StringVar`'s `.get()`.

"%W" is the name of the widget.

"%v" is the current value of `validate`.

So Imagine that I would be happy with any string that does not contain an @ sign. I would need to add a function and a registration step, and change the call to the `Entry`'s constructor:

```
1 >>> def ok(newval):
2 ...     return newval.find("@") < 0
3
```

```

4 >>> validator = win.register(ok)
5
6 >>> inw = tk.Entry(win, width = 20,
7 ...           textvariable = svin,
8 ...           background = "white",
9 ...           validatecommand = (validator, "%P"),
10 ...          validate = "all")

```

This only results in nothing at all happening when the user types an @, no explanations or warnings are given to the user. You would need extra code in the ok function to make some easily seen explanation appear somewhere in the gui.

I think you would be better of forgetting about this feature, and just perform validation on the final result of all the user's activities when the "do it now" button is pressed.

An `Entry` also has a lot of methods. When a cursor position is given, 0 means before the first character, 1 means between the first and second characters and so on. When a character position is given, it is the normal Python way, 0 means the first character. And don't forget about the universal methods: `configure` and so on.

`e.show(string)`

This is used for password entry fields. Whenever the user types anything into the `Entry`, every character typed is just show as though the first character of the given string had been typed instead. Of course, the value that you retrieve from the `StringVar` contains the real characters that were actually typed.

`e.insert(pos, s)`

Insert the string `s` into the string and the display, just before character `pos`.

`e.delete(first, last)`

Delete characters from the string and the display, beginning with the `first`<sup>th</sup> character, and ending *before* the `last`<sup>th</sup>. Default for `last` is all the way to the end.

`e.icursor(pos)`

Move the cursor to character `pos`.

`e.select_clear()`

Deselect everything. The selection is the usually highlit range of characters that the user has dragged the mouse across prior to copying or deleting.

`e.select_range(first, last)`

Select everything starting from the `start`<sup>th</sup> character and ending just before the `last`<sup>th</sup> character.

`e.select_present()`

True or False: is there anything selected at the moment?

```
e.selection_get()
```

Return the currently selected substring.

## 45. Binding to mouse and keyboard events

The following is written with the example of an `Entry` object in mind, but all of this applies equally to all other `tkinter` objects.

Users don't expect to have to press a button in order to have their input accepted. Pressing enter is usually enough. `tkinter` allows keypresses to be bound to functions so that the function is called whenever that key is pressed, so long as a particular widget has focus. It is easy to do. I could just add

```
1 >>> win.bind("<Return>", doit)
```

immediately after creating the window, then the `doit` function would be called if anything in the window has focus when enter is pressed. It is almost certainly better to be more precise with this instead:

```
1 >>> inw.bind("<Return>", doit)
```

right after the `Entry` `inw` is packed. That way `doit` is only called if that particular `Entry` has focus when enter is pressed. You can have a different function attached to enter for each of a large number of `Entry` objects.

One more change would be required. When a function is called as a result of a bound key being pressed, it is given a parameter, a `KeyPress` event object that says exactly what the triggering event was. But when a function is called because it is a `Button`'s command callback, no parameter is provided. Just give `doit` a parameter with a default value so both cases will be valid.

To bind a keypress other than enter, you can usually just provide that character as the string, as in

```
1 >>> win.bind("a", doit)
```

The exceptions are for space and `<`, which become `<space>` and `<less>`. Note that `Return` must have a capital letter while `space` and `less` must not. If you use `<Key>` then any key, even a shift key, will trigger the function call, and you'll have to look at the event parameter to see which one it was.

These are "all" the keypresses you can bind, but just by example of a few from each category. Note that capitalisation is totally inconsistent but you have to get it right.

```
"<Return>", "a", "q", "Q", "3", "7", "#", "(", "$", "\", "<Control-u>",  
"<Control-slash>", "<Control-backslash>", "<Control-Key-1>",  
"<Control-Key-7>", "<Tab>", "<space>", "<less>", "<BackSpace>",  
"<Delete>", "<Escape>", "<F1>", "<F7>", "<Alt-h>", "<Alt-q>",  
"<Left>", "<Up>", "<End>", "<Home>", "<Prior>", and "<Next>".
```

Note: `<Prior>` and `<Next>` are `tkinter`'s names for Page Up and Page Down. `Control-Key-n` refers to the non-standard combination of typing a digit while holding down the control key.



You can also bind to some mouse events. `Button-1` always refers to the left mouse button, but the right is sometimes `2` and sometimes `3`. On this computer, "`<Button-3>`" is for a right-click, "`<B1-Motion>`" is for dragging the mouse while keeping the left button pressed. For the details of a mouse drag you need to look at the value (let's call it `e`) passed to your callback function. `e.x` and `e.y` are the coordinates of the mouse pointer at the time the event was recorded. You will get a lot of these events while the mouse is being moved, so you can trace its path quite well. `e.type` will be `tkinter.EventType.Motion`.

In any given component, each event can only be bound to a single callback. A subsequent `bind` for that event to a different callback will cancel the original bind. To cancel a bind deliberately, use:

```
1 >>> inw.unbind("<Return>")
```

In fact, all `tkinter` components support `bind` and `unbind`, but that is of limited use for some of them. `Canvasses` and `Labels` can not take keyboard focus, so binding keyboard events for them has no effect. But you can still bind mouse events for them, to detect clicks and so on. The list of mouse events is:

"`<Button-n>`" or just "`<n>`":

The  $n^{\text{th}}$  mouse button is clicked.

"`<ButtonPress-n>`"

"`<ButtonRelease-n>`"

The two components of a click. Pressing and clicking can not be distinguished on my Windows PC.

"`<Double-Button-n>`"

The  $n^{\text{th}}$  mouse button is double-clicked. Also not distinguishable from clicks on my PC.

"`<Triple-Button-n>`"

The  $n^{\text{th}}$  mouse button is triple-clicked. Has no effect at all on my Windows PC.

"`<Bn-Motion>`"

The mouse is dragged while its  $n^{\text{th}}$  button is held down.

"`<Enter>`"

The mouse cursor has entered the widget's space. Clicks not necessary.

"`<Leave>`"

The mouse cursor has left the widget's space.

There are also a few other kinds of event:

"`<Configure>`":

The widget's size or position has changed.

"`<FocusIn>`":

Keyboard focus has been gained by this widget, or if it is a container, possible one of the widget that it contains.

"`<FocusOut>`":

This widget has lost keyboard focus.

Individual Canvas items may also have events bound to callback functions:

`c.tag_bind(items, event, function = None, add = None)`

`event` may be any of the mouse events that are described a bit later in the sub section for an `entry` (sometimes there is no order of presentation that really works) such as "`<Button-1>`" for a left mouse click. Every time the given event occurs directly on top of any of the items, the callback function will be called. "Directly on top of" is modified by the `closeenough` Canvas option. The function should have exactly one parameter, and it will be an object describing the event. Normally this function replaces any function(s) already bound to this event and this item, but if `add` is "+", it becomes an additional callback for the same event. The return value is a string that can be used later with the `tag_unbind` method.

`c.tag_unbind(items, event, funcId = None)`

The parameters are as for `tag_bind`, `funcId` is the string that `tag_bind` returned. The callback function for these items and this event is removed. If `funcId` is `None`, all the callbacks are removed.

## 46. Text

A `Text` object is a vast extension of an `Entry`. It is multi-line, multi-font, multi-background, images can be included, text editing operations are available. A `Text` object does not provide buttons or menus allowing the user to command these things, the programmer has to provide all such things. You can do that by key bindings as described for `Entry` objects, or by creating `Buttons`, or by providing menus, which is still to come.

`background`, `bg`, `foreground`, `fg`, `textvariable`, `font`, `relief`, `borderwidth`, `border`, `bd`, `cursor`, `padx`, `pady`, `insertwidth`, `insertbackground`, `insertontime`, and `insertofftime`.

Exactly as they are for `Entrys` and earlier things.

`height` and `width`

Are measured in characters, not pixels.

`state` = "normal" or "disabled"

The same as for everything else.

`tabs` = tuple of list of strings

The strings give the positions of the tab stops, they are in the usual distance format, "1.5i" for inches, "1.5c" for centimetres, "1.5m" for millimetres, and "1.5p" for points. Each position is optionally followed by one of "left", "right", "center", or "numeric" to specify the kind of tab. The first three mean what they normally mean in word processing. "numeric" means that the decimal points in the text immediately following the tab insertion will be aligned at the tab position. The decimal point is simply the

first "." encountered. If there is no decimal point then the entire string is deemed to be before the decimal point.

`wrap = "char", "word", or "none"`

If a line is too long for the given width, it will be split at the first character that doesn't fit, the previous piece of white space, or not at all, respectively. "word" is the default.

`selectbackground = colour`

`selectforeground = colour`

When a region of text is selected, either by the user dragging the mouse across it, or as a result of a method call, these colours will be used in displaying it.

`spacing1, spacing2, or spacing3 = size, defaults are 0`

Extra space above and below each line. 1: space above, 3: space below. If a long line is wrapped, 1 only applies above the whole thing and 3 only below the whole thing. 2 is the extra internal vertical space within wrapped lines.

And of course a `Text` has all the universal methods: `configure` and so on.

There is absolutely nothing to stop the contents of a `Text` from being bigger than its height and width specified. The display will automatically move around to ensure that the input cursor remains visible after any user activity (inserts, deletions, arrow keys, etc). You can add scrollbars to give the user more control, and that is covered soon, but there is a subclass called `ScrolledText` that automatically adds a vertical scroll bar, that is coming later in this subsection. The program can itself cause the text to scroll with these three methods. The first one uses an index, which is a way to specify a position in a `Text`'s contents, and is described next.

`t.see(index)`

Does whatever scrolling is necessary to make the text at the index visible.

`t.xview("scroll", n, "units" or "pages")`

Moves the contents horizontally by a distance of `n`. What `n` means is determined by the second parameter, "units" means `n` characters, "pages" means `n` times the `Text`'s width. Calling that a "page" makes sense for vertical movement. If `n` is positive the text moves to the left (We normally expect left to be negative and right to be positive, but this method is really intended to be used automatically by a `Scrollbar` widget, and the text moves left when the scrollbar moves right), and if `n` is negative it moves right.

`t.xview("moveto", pos)`

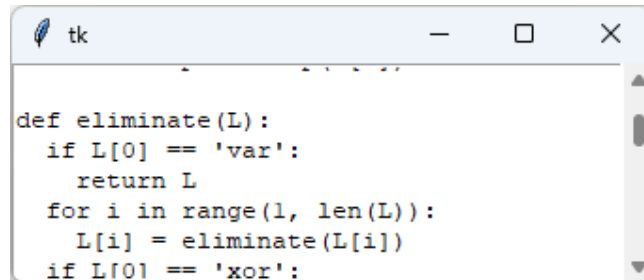
`pos` should be a float between 0 and 1. Moves the contents horizontally so that position `pos` is as close to the left edge as possible. `pos = 0` means that a scrollbar would be all the way to the left, `pos = 1` means that a scrollbar would be all the way to the right.

```
t.yview(...)
```

Is the same as the two `xviews` except that the movement is vertical.

This small example uses a `ScrolledText` rather than a plain `Text` widget, but they are both used in exactly the same way. The only difference is that a `ScrolledText` automatically adds a scroll bar to the right. The `insert` method used here is explained in the next subsection. Note also that `ScrolledText` is imported from `tkinter.scrolledtext` whereas `Text` is imported from plain old `tkinter`.

```
1 >>> from tkinter.scrolledtext import ScrolledText
2 >>>
3 >>> win = Tk()
4 >>> st = ScrolledText(win, width = 40, height = 7)
5 >>> st.pack(fill = "both", side = "left", expand = True)
6 >>> fi = open("boolean.py", "r")
7 >>> st.insert("1.0", fi.read())
8 >>> fi.close()
```



### i. Index - positioning, inserting, etc

An index is used to refer to a position within the text and images. It is always a string. Indexes are used with many of a `Text`'s methods, such as `w.get(index1, index2)` and `w.delete(index1, index2)` which retrieve or erase the text between the two indexes. If you have images within the text, they are considered to be single characters.

"12.15":

Just before the character 15 on the twelfth line. Lines start counting from 1, but characters start counting from 0, so "4.0" is the beginning of the fourth line, "1.0" is just before the first character of all. It is not an error for a number to be too big. If the line number is beyond the end of the text, it is taken as meaning the end of the text. If the character number is beyond the end of its line, it is taken as meaning the end of that line.

"4.end":

Just after the last character on the fourth line.

"end":

Just after the very end of all the contents, the opposite of "1.0".

"insert":

The position of the cursor. Not the ordinary cursor that follows the mouse everywhere, but the vertical line insertion cursor specific to this `Text`.

"current":

The closest index to the current position of the mouse. A character has to be completely left of the mouse cursor in order to be included.

"@x,y":

x and y must be numbers. The closest index to position (x, y) within the `Text` object, measured in pixels.

"sel.first":

The index of the first character in the current selection. Exception if there isn't currently a selection.

"sel.last":

The index of the last character in the current selection.

"abc":

You can create things called marks at any position in the text. Marks have names. If abc is the name of a mark, then "abc" means the position of that mark.

"abc.first":

As well as marks, there are also tags. A mark represents a single position, but a tag represents a whole region of text. If abc is the name of a tag, then "abc.first" is the position of the first character within that tag.

"abc.last":

The position of the last character within a tag.

Index strings may be extended by adding any number of extra strings in the following formats. They are interpreted as modifiers to a position in left to right order. "4.7 + 3 lines + 8 chars" is (usually) equivalent to "7.15". Only usually because "+ n chars" doesn't stop if it reaches the end of a line, it just moves on to the next one.

"+ n chars" or "+nchars"

"- n chars" or "-nchars"

"+ n lines" or "+nlines"

"+ n lines" or "+nlines"

The obvious meanings, given the little example above.

"linestart"

"lineend"

The very beginning or very end of the line we are on now.

"wordstart"

Just before the beginning of the word that contains the current position. A word is considered to be any unbroken sequence of letters, digits, and underlines. If the current position isn't within any such thing, then it just moves back by a single character.

Indexes are operated on by various methods of the Text object:

`t.index(index)`

However the index is specified, it is converted to the first form, "line.character" and that string is returned.

`t.get(index)`

`t.get(index1, index2)`

The first form returns the character immediately after the given index. The second form returns a string containing all the characters that come after `index1` and before `index2`, so `w.get("1.0", "1.3")` return the first 3 characters of the first line. `\ns` are included if the region spans more than one line. Images are ignored in constructing the string.

`t.delete(index)`

`t.delete(index1, index2)`

The indexes describe a range of text in exactly the way they do for `get`. The range of text is deleted.

`t.insert(index, string)`

The string is inserted at exactly the position of the index.

`t.compare(index1, relationship, index2)`

Relationship may be any one of "=", "!", "<", ">", "<=", and ">=". Returns `True` or `False` depending upon whether or not the given relationship holds between the relative positions of `index1` and `index2`.

`t.bbox(index)`

Short for bounding box. Characters are displayed so that their bounding boxes are pressed up against each other, usually with no space between them, and always with no overlap. Returns a tuple giving the position and size, all in pixels, of the bounding box for the character following the index. If the index is at or beyond the end of a line, the box returned describes the unused portion of the line, from the right edge of the last character to the `Text`'s own right edge.

The bounding box is in the form  $(x, y, w, h)$ ,  $x$  and  $y$  are the co-ordinates of the top left corner,  $w$  and  $h$  are the width and height.

`t.dlineinfo(index)`

Exactly the same as `bbox`, except that it is the bounding box for the whole line that contains the given index.

`t.search(string, index, ...options...)`

Searches for a match for the string, starting at the given index. It returns an index in its most basic form "line.character" showing where the match starts or an empty string if there is no match. All of the options are optional, they are:

`nocase = 1`

The search will be case insensitive.

`stopindex = index`

Limits the scope of the search, it will not go beyond this index.

`exact = True or False, True is the default.`

Requires an exact match for the string, it is the opposite of `regexp`.

`regexp = True or False, False is the default.`

Treats the string as a regular expression and accepts any match for it. tkinter's regular expressions are not as powerful as Python's own regular expressions, the only special characters are `.`, `^`, `[ ]`, `( )`, `|`, `*`, `+`, and `?`, and nothing has any special meaning inside `[...]`.

`forwards = True or False, True is the default.`

Searches from the index towards the end of the text.

`backwards = True or False, False is the default.`

Searches backwards from the index towards the beginning of the text.

`count = IntVar.`

The number of characters from the text that matched the pattern will be stored as its value.

## ii. Mark - automatically updated position reference

A mark gives a name to a position within the text, and it is updated automatically. If the text before a mark is inserted or deleted, the mark moves forwards or backwards to keep its position between the same two characters.

If the text around a mark is deleted, the mark remains valid, and represents the new position where the deleted text used to be.

If text is inserted at the exact position of a mark, what happens depends on the value of the mark's "gravity". Gravity may be either "left" or "right". "left" means the mark would move to the left of the inserted text, and "right" means it would move to the right of the inserted text. The default gravity is "right".

Marks do not have much of an existence of their own, they are entirely controlled by the `Text`'s own methods:

`t.mark_set(name, index)`

Creates or modifies a mark at the position given by index. The name may be any string so long as it doesn't include any ' .' or whitespace characters.

`t.mark_unset(name)`

Deletes the named mark.

`t.mark_gravity(name, "left" or "right")`

`t.mark_gravity(name)`

The first form changes the gravity of a mark, the second form returns it.

`t.mark_next(index)`

`t.mark_previous(index)`

Return the name of the next mark either following or preceding the given index. If the index starts with the form "name" (i.e. the index is actually a mark), it is not an eligible return value, the next before or after it will be returned. In other cases, if the given index exactly matches the position of a mark, then that mark will be returned. The value is an empty string if there is no mark meeting the requirements.

`t.mark_names()`

Returns a tuple of all the mark names currently in existence. It will always at least include "insert" and "current", which always count as mark names corresponding to the same-named indexes.

### iii. Tag - control over multiple regions

As stated a little while ago, a tag describes a region of text, but it is more than just two indexes. Tags do not have to describe contiguous areas, they represent any number of contiguous regions of text, each of which could be described by a start index and an end index. Tags can overlap, any region of text can be covered by as many tags as you want.

The purpose of a tag is not really just to describe a region of text, but to give you control over it. You can set any of the properties: colour, font, justification, etc, all at once. If two tags overlap and demand different settings (perhaps a run of characters is within two tags, one of them sets it to orange and the other sets it to blue) there is a "tag stack" that resolves the conflict. Essentially, newer tags outrank older tags. You can modify the tag stack to change the order.

If any of the text is selected (usually because the user dragged the mouse across it while keeping the left button down), there will automatically be a tag named "sel" that represents the selected region. The programmer can cause text to be selected by using the `tag_add` method on "sel".

The `Text` methods that control tags are:

`t.tag_add(name, index1, index2)`



The region between the two indexes becomes part of the named tag. If no such tag exists, it is created. The second index is optional, if it is absent, then just the single character immediately after the first index is taken.

`t.tag_remove(name, index1, index2)`

The same parameters as `tag_add`. the region of text is no longer controlled by the named tag.

`t.tag_delete(name)`

The tag is removed everywhere from the entire `Text`, and any settings of options that it imposed are undone.

`t.insert(index, string)`

This is primarily a method for working with an index, but it has an effect on tags. Tags define a region of the text, so if the index point is within a tag, the entire inserted string will also be covered by that tag. There is an optional third parameter which must be a tuple of strings, the strings being tag names.

The inserted text will all belong to all of the named tags. If a tag with that name does not already exist, it will be created.

`t.tag_config(name, option = value, option = value, ...)`

For all the text controlled by the named tag, the given options are set to the given values. The applicable options are the familiar ones: `background`, `foreground`, `font`, `justify`, `underline`, `tabs`, `borderwidth`, `spacing1`, `spacing2`, `spacing3`, and `wrap`.

Plus some new ones:

`lmargin1`, default 0

indentation for first lines.

`lmargin2`, default 0

indentation for successive lines

`rmargin`, default 0

space to be left between the end of a line and the `Text`'s right edge

`offset`, default 0

distance characters must be raised or lowered compared to their neighbours (for superscripts and subscripts)

`overstrike`, default 0

`overstrike = 1` means a horizontal line is drawn through the middle of the text.

`t.tag_config(name)`

Returns a dictionary of all the options that have been set for this tag and their values.

`t.tag_cget(name, option)`

Returns the value of the option that has been applied to this tag.

`t.tag_bind(name, event, callback)`

If the given event occurs in an area controlled by the named tag, the callback function will be called with one parameter that describes the event. The events that can be bound are just as they were for `Entry` objects, "`<Button-1>`" for a mouse left-click, "`w`" for a lower case `w` being typed, etc.

`t.tag_unbind(name, event)`

Undoes the effect of `tag_bind`.

`t.tag_names()`

Returns a tuple of the names of all known tags in this `Text`.

`t.tag_names(index)`

Returns a tuple of the names of all of the tags that control the text at `index`.

`t.ranges(name)`

Returns an even-length list of indexes giving the entire portion of the `Text` that is controlled by the named tag. The indexes come in pairs, the first of each pair says where a subregion begins, the second says where it ends.

`t.tag_nextrange(name, index1)`

`t.tag_nextrange(name, index1, index2)`

Searches forward from `index1` to the end of the `Text`, looking for the beginning of a region controlled by the named tag. It returns a tuple of two indexes: that region's beginning and end, or a zero-tuple if none is found. If `index2` is provided, the search ends there instead of at the end of the `Text`.

`t.tag_prevrange(name, index1)`

`t.tag_prevrange(name, index1, index2)`

Exactly the same as `tag_nextrange`, except that it searches backwards from `index1` to the beginning of the `Text`. It is still only searching for beginnings of regions.

`t.tag_raise(name, aboveThis = None)`

If `aboveThis` is `None`, the named tag goes to the top of the priority stack. Otherwise `aboveThis` must be another tag name, and the first named tag's priority is raised until it is just above the second. If the change in priority results in any conflicts being resolved differently, the display is updated accordingly.

`t.tag_lower(name, belowThis = None)`

Just like `tag_raise`. If `belowThis` is `None`, the named tag goes to the bottom of the priority stack. Otherwise `belowThis` must be another tag name, and the first named tag's priority is reduced until it is just below the second.

#### iv. Images

A `PhotoImage`, as described in the `Canvas: Image` subsection, can also be inserted into the text. It is treated as a single character.

`t.image_create(index, image = PhotoImage, options)`

The image appears in the text at the `index`'s position. There are only a very few options:

`name = string`

Gives the image a name. You can find an image's position if you know its name. If you do not provide a name, `tkinter` will make one up, and return it as the result of `image_create`.

`padx` and `pady`

Have their usual meanings..

`align = "top", "bottom", "baseline", or "center",`

The height of a line is always the height of the tallest thing within it plus any extra spacing caused by `spacing1/2/3`. Usually an image will be taller than anything else, in which case `align` does not come into effect. But if you insert two images of different heights, one of them will be shorter than the line that contains it. `align` says what its vertical position should be in this case. "top" aligns the image's top with the line's top, "center" aligns centre with centre, and "bottom" aligns bottom with bottom. "baseline" aligns the image's bottom with the line's baseline. The baseline is the bottom of any normal character that has no descender such as p or q.

`t.index(imagename)`

This is an extra use of the `index` method in the `Text/Index` subsection. Returns the position of the image in the usual "line.character" string format.

`t.image_configure(index, option = value, ...)`

You can only refer to an image by its index, which is for the position just to the left of the image. This method lets you change any of the very few options given to `create_image`. If you do not provide any `option = value` pairs, it returns a dictionary of all of them.

`t.image_cget(index, option)`

Returns the value of the given option.

`t.image_names()`

Returns a tuple of the names of all the images in this `Text` object.

## v. Window - inserting widgets in text.

Images are not the only non-text thing that can appear in a `Text` object. Any `tkinter` widget can go in there, even a container with many other widgets inside

it. Such objects occupy what is called a window in the text. Like images, windows are treated as single characters.

`t.window_create(index, options)`

Strangely, `window_create` does not allow you to provide a name, but if you ever want to find a window, you'll need to know its name. To handle this, you must give a name to the widget that the window is based on. When you create the widget, just use the `name = "xxx"` option. If the widget's name is "xxx" then the window's name will have a dot added at the beginning: ".xxx".

Also remember that when you create a widget, the first parameter to its constructor must be the container that will contain it. But these widgets are not going into a real container. I've found that giving `None` as that first parameter, or the main window itself, both work well.

These are the possible options. You must provide either `window` or `create` but not both.

`align, padx, and pady`

The same as for an image.

`stretch = int, the default is 0.`

If `stretch` is set to 1 and the widget is shorter than the line that contains it, then `align` will be ignored and the widget will be stretched vertically to fill the line's height instead.

`window = widget`

This is the widget that is to be embedded in the text.

`create = function`

This parameterless function will be called and the widget that it returns is the one that will be embedded in the text.

`t.index(imagename)`

Just like for images.

`t.window_configure` and `t.window_cget`

Are exactly the same as `t.image_configure` and `t.image_cget`.

`t.window_names()`

Returns a tuple of the names of all the windows in this `Text` object.

## vi. Undo and Redo

There are three more options available to the `Text` object's constructor, they are `undo` (default `False`), `maxundo` (default `-1`), and `autoseparators` (default `True`). They control the object's ability redo and undo editing operations.

If `undo` is left at `False`, there is no ability to undo operations, and nothing else in this subsection will work. When `True`, everything that makes a change to the text, whether it was done by the user typing or deleting or by the program using the object's methods, is recorded on a stack. `maxundo` says the size of the stack, `-1` means unlimited. If `maxundo` is exceeded, the oldest entries are lost.

`t.edit_undo()` and `t.edit_redo()`

Reverses the effect of the last changes, how many will be explained very soon. Those changes are not completely discarded immediately. As soon as another change is made after an `edit_undo`, those changes are discarded. If no further change has been made, an `edit_redo` will reverse the effect of the last `edit_undo`, restoring the effects of all the undone editing operations.

`t.edit_separator()`

Adds a special entry, called a separator, to the stack. An `edit_undo` will always undo all the operations on the stack, back to (and including) the last separator. The system will not allow two consecutive separators on the stack.

If `autoseparators` is `True`, the system automatically adds a separator after each closely related sequence of operations. What counts as closely related is a little inconsistent, at least on this Windows PC. Any sequence of `t.inserts`, even to totally different parts of the text, and any sequence of `t.deletes` to anywhere at all, are considered to be closely related. When the user is typing or deleting, only consecutive operations are closely related. Type something somewhere, move the cursor, and type something somewhere else becomes two sequences with a separator between them.

`t.edit_modified()`

The system records whether or not the text has been changed in a special `Bool` flag. Every time any change is made, the flag is set to `True`, but `edit_undo` and `edit_redo` are taken into account, so the flag can go back to being `False`. This method returns the value of that flag. If you've been editing a file and find that the flag is `False`, there is no need to re-save the results.

`t.edit_modified(True or False)`

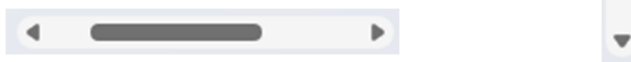
Explicitly sets the value of that flag. If you initialise a `Text` object with the contents of a file, you probably want to consider that to be the original unmodified state, but the act of inserting the file's contents would have set the flag to `True`. This method lets you choose what is considered to be the unmodified state.

## 47. Scrolling

There are widgets that represent horizontal or vertical scroll bars, and other widgets that could have more content than can fit inside a reasonably sized one

(such as Canvasses and Texts) have ways of being controlled by Scrollbar widgets.

Everyone is perfectly familiar with scroll bars, but we need to get tkinter's terminology for them sorted out, so here are two of them.



The dark bit in the middle, that you move to make big changes in position is called the slider. We generally expect the size of the slider to be proportional to the portion of the document we can currently see. The pale areas to the left and right, or above and below, that you typically click on to move by approximately a page are called troughs. The top or left one is trough 1, the bottom or right one is trough 2. The triangles at the ends, that we click on to move by a fairly small amount are called arrow 1 and arrow 2.

The constructor follows the usual pattern for widgets:

```
sb = tk.Scrollbar(window, option = value, ...)
```

and it has many of the familiar options. These ones are exactly as we are used to them being: `background` (or `bg`), `borderwidth` (or `border` or `bd`), `cursor`, `relief`, `activebackground`, and `activerelief`, and `command` provides the callback to be used whenever there is any movement. `background` (or `bg`) is taken as the colour of the slider and the arrows. The `active...` options do not work on my Windows PC.

The new options are:

`orient = "horizontal" or "vertical" (default "vertical")`

With the obvious meaning.

`width = size`

The narrow dimension of the whole `Scrollbar`. For a vertical one there is no question about what `width` should mean. For a horizontal one, we might think this should be called `height`, but it isn't.

`troughcolor = colour`

The colour for the troughs.

`jump = 0 or 1 (default 0)`

The usual behaviour is that after even the slightest movement of the slider, the callback is called so that the display can be updated in time with it. If `jump` is 1, nothing happens until the slider is in its new position and the mouse has been released.

`elementborderwidth = size (default -1)`

The width of the borders around the arrows and slider. The illustrations don't show any border because that isn't the current Windows style, but

blockier guis often present the slider as a rectangle with raised borders. `-1` means that it should be the same as `borderwidth`.

```
repeatdelay = milliseconds (default 300)
repeatinterval = milliseconds (default 100)
```

If the mouse button is held down in one of the troughs for longer than `repeatdelay`, the callback will be called again, and then again and again after every `repeatinterval`.

To make a `Scrollbar` work, it must be configured to know about the component that it is controlling, and that component needs to be configured to know about the `Scrollbar`.

Everything that can be scrolled has `xscrollcommand` and `yscrollcommand` options and `xview` and `yview` methods.

If you create the scrollable thing first, then set your `Scrollbar`'s `command` option to be the scrollable's `xview` (for horizontal) or `yview` (for vertical) method, then configure the scrollable's `xscrollcommand` or `yscrollcommand` to the `Scrollbar`'s `set` method.

If you create the `Scrollbar`s first, then it is the other way round. When you create the scrollable widget, set its `xscrollcommand` and/or `yscrollcommand` options to the appropriate `Scrollbar`'s `set` method, then configure the `Scrollbar`'s `command` option to be the scrollable's `xview` or `yview` method.

Here is the whole thing:

```
1 >>> win = tk.Tk()
2 >>> txt = tk.Text(win, width = 50, height = 10, wrap = "none")
3 >>> txt.grid(row = 0, column = 0)
4 >>>         # use txt.insert("1.0", ...) to give the Text its content
5 >>> hsb = tk.Scrollbar(win, orient = "horizontal",
6 ...                   width = 20, command = txt.xview)
7 >>> hsb.grid(row = 1, column = 0, sticky = "ew")
8 >>> vsb = tk.Scrollbar(win, orient = "vertical",
9 ...                   width = 20, command = txt.yview)
10 >>> vsb.grid(row = 0, column = 1, sticky = "ns")
11 >>> txt.configure(xscrollcommand = hsb.set)
12 >>> txt.configure(yscrollcommand = vsb.set)
```

If you don't set the `Text`'s `wrap` to `"none"` you will never get an opportunity to use the horizontal `Scrollbar`. If you don't use `sticky` when you grid the `Scrollbar`s they will be so small that they can't be moved.

The slider can also be moved by the program through the following methods. You should almost certainly not do this unless you also move the contents by the corresponding distance with the `Text`'s `xview` and `yview` methods. Even then,

there isn't much point: Calling the `Text`'s `xview` or `yview` method automatically moves the slider to the correct position.

`sb.get()`

Returns the tuple `(L, R)` describing the slider's position and size. `L` and `R` are both floats in the range 0 to 1, where 0 means the left or top and 1 means the right or bottom of the `Scrollbar`. `L` gives the left or top edge of the slider, and `R` give the right or bottom.

`sb.set(L, R)`

`L` and `R` should be exactly as described for `get`. The slider's size and position are changed to match them.

`sb.delta(xchange, ychange)`

`xchange` and `ychange` represent a desired movement in pixels. It returns a float between 0 and 1 which is the amount that would need to be added to the slider's `L` and `R` values to cause that movement. There really should not be two parameters: a horizontal `Scrollbar` completely ignores the value of `ychange`, and a vertical one completely ignores `xchange`.

`sb.fraction(x, y)`

`x` and `y` are a position inside the scrollable thing, measured in pixels. Returns a float in the range 0 to 1, which is the value that the slider's `L` should be set to in order to move that position to the left or top.

`sb.identify(x, y)`

`x` and `y` pixel coordinates relative to the `Scrollbar`'s own top left corner. Returns `"arrow1"`, `"trough1"`, `"slider"`, `"trough2"`, or `"arrow2"` depending on which of those is at that position. If no part of the `Scrollbar` is at that position, it returns the empty string.

`sb.activate()`

`sb.activate("arrow1", "slider", or "arrow2")`

This method does not work on my Windows PC. The first form is supposed to return either `"arrow1"`, `"slider"`, or `"arrow2"`, if the mouse is directly above one of those things, or the empty string if it isn't. The second form is supposed to put that named part into the active state, so it will be coloured appropriately. Why would they leave out `"trough1"` and `"trough2"`?

`Texts` are not the only widgets that can have `Scrollbars` attached to them. A `Canvas`, a `Listbox`, and a `Treeview` can too. `Listbox` and `Treeview` will be covered soon. They all have a `xscrollcommand`, `yscrollcommand`, `xview`, and `yview` that work exactly the same as they do with a `Text`.

A `Canvas` provides three extra options:

`scrollregion = (L, T, R, B)`

The other three scrollable widgets have a natural size determined by their contents, so how far you are allowed to scroll in any direction is perfectly



clear. But a programmer could potentially draw anywhere on a Canvas, so the scrollable region can be controlled. All four values in the tuple are positions measured in pixels, being the Left, Top, Right, and Bottom edges of the region that can be scrolled through. One would normally expect L and T to be 0, and R and B to be the width and height of the imaginary piece of drawing paper that we are scrolling over, but that is not an actual restriction.

The numbers really do describe the imaginary paper, but L and T can be negative if you want. The default position when everything is first displayed is that your paper's (0, 0) will be at the top left of the visible area.

```
xscrollincrement = size
yscrollincrement = size
```

size is in pixels if it is an int, but it may also be in the string format, "0.25i" for a quarter of an inch for instance. Setting these makes the scrolling jumpy. If you very slowly move the slider, nothing will happen for a while, but then all of a sudden the viewing area moves by the specified distance. The visible portion of the paper can only start at an exact multiple of size from its top or left edge.

Additionally for Canvasses and Texts only, there are methods for what is called fast scrolling. The idea is that when the user drags the mouse across the canvas while keeping a mouse button pressed, the canvas should scroll as though the paper were being pulled in the same direction. These two methods enable that behaviour:

```
w.scan_mark(x, y)
w.scan_dragto(x, y, gain = 10)
```

scan\_mark should be called when the chosen mouse button is pressed, and scan\_dragto is called if mouse movement is detected while the button is still pressed. It isn't possible to directly ask whether a mouse button is up or down, but motion events are only generated for mouse movements while a button is held down. So the appropriate mouse events should be bound to callback functions that call these methods. If w is a Canvas or Text:

```
1 >>> def pressed(evt):
2 ...     w.scan_mark(evt.x, evt.y)
3
4 >>> def dragged(evt):
5 ...     w.scan_dragto(evt.x, evt.y, 3)
6
7 >>> w.bind("<ButtonPress-1>", pressed)
8 >>> w.bind("<B1-Motion>", dragged)
```

The gain parameter says how fast the paper should move, it has no particular units. I find that the default of 10 is far too reactive, and 3 is more comfortable.

## 48. Checkbutton

A Checkbutton is the familiar simple mechanism for allowing the user to make a binary yes/no choice. Here are three of them:



In a very simple situation, they are very simple to set up:

```
1 >>> from tkinter import *
2 >>> iv1 = IntVar()
3 >>> cb1 = Checkbutton(win, variable = iv1, text = "Measles")
4 >>> cb1.grid(row = 0, column = 0, padx = 10)
5 >>> iv2 = IntVar()
6 >>> cb2 = Checkbutton(win, variable = iv2, text = "Chicken pox")
7 >>> cb2.grid(row = 0, column = 1, padx = 10)
8 >>> iv3 = IntVar()
9 >>> cb3 = Checkbutton(f, variable = iv3, text = "Plague")
10 >>> cb3.grid(row = 0, column = 2, padx = 10)
```

From now on, I'm going to assume that `*` for `tkinter`'s import, that way I won't have to type `tk.` in front of so many things, and I can fit a little more on a line.

Checkbuttons are controlled by an `IntVar`, whenever it is selected or deselected the `IntVar` is automatically changed (by default 0 is for off, and 1 is for on). Each `Checkbutton` takes its initial state from the original value of its `IntVar`, so if I wanted `Chicken pox` to be initially selected, I would have typed `iv2 = IntVar(value = 1)`. The program may turn the `Checkbutton` on and off by itself, either by using the `IntVar`'s `set` method, or by using the `Checkbutton`'s `select`, `deselect`, and `toggle` methods.

A `Checkbutton`'s description is usually given by the `text` option, as in the example, but you can instead set the `textvariable` option to a `StringVar`. Then its description can be changed easily whenever desired.

As with a `Button`, the description can be a string (set `text` or `textvariable`), or a graphic (set `image`), or both if you set `compound`.

These options are exactly the same as for `Buttons`: `background` or `bg`, `foreground` or `fg`, `borderwidth` or `border` or `bd`, `font`, `justify`, `cursor`, `padx`, `pady`, `relief`, `overrelief`, `state`, `underline`, `wraplength`, `anchor`, `disabledforeground`, `activebackground`, and `activeforeground`.

`image = PhotoImage`

The same as for a `Button`, this provides an image that will be used instead of text.

`compound = "top", "bottom", "left", "right", or "center"`

The same as for a `Button`, allows an image and some text to both appear.

```
selectimage = PhotoImage
```

This only works if you have provided an image too. When the Checkbutton is on, this image will appear instead.

height specifies the number of lines of text but even without it \ns are treated correctly. width is measured in characters not pixels.

The documentation claims that selectcolor specifies the colour that the widget should have when it is on, but at least on my Windows PC it just specifies the colour of the little box regardless of whether it is on or off.

```
indicatoron = 0
```

When set to 1, the Checkbutton will not have the usual little box to show its state, empty for off and ✓ for on. Instead, the whole thing will look just like a Button, with (by default) a raised border when it is off, and a sunken border when it is on. Also, selectcolor starts to work as the documentation says: the entire background of the Checkbutton changes to selectcolor when it is on.

command should be a parameterless callback function, it is called every time the Checkbutton is turned on or off by a mouse click, changes made by the program do not cause a callback. Callbacks usually have an event parameter so you can tell what happened. In this case you just have to use the IntVar's get method. The IntVar will always be changed before the callback happens.

```
onvalue = 1
```

```
offvalue = 0
```

These provide the values that the IntVar will be set to to indicate whether the Checkbutton is on or off.

```
overrelief = "raised", "sunken", "groove", "ridge", or "flat"
```

The relief around the Checkbutton will be changed to this style whenever the mouse is hovering over the Checkbutton. Remember that relief isn't possible unless you also have a border of a few pixels. Unlike most things to do with mouse hovering, this does work under Windows.

```
offrelief = "raised", "sunken", "groove", "ridge", or "flat"
```

The relief around the Checkbutton will be set to this style whenever the Checkbutton is off. This does not work on my Windows PC.

The Checkbutton's extra methods are:

```
c.select()
```

```
c.deselect()
```

Turn the Checkbutton on or off.

```
c.flash()
```

For about a second or maybe less, the Checkbutton rapidly switches between its current colour scheme and the alternative given by

activebackground and activeforeground. In this case, the active... settings do work under Windows.

`c.invoke()`

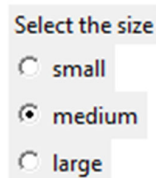
Causes the callback function to be called.

`c.toggle()`

If the Checkbutton is on, it is turned off. If it is off, it is turned on.

## 49. Radiobutton

A Radiobutton is a lot like a Checkbutton, most of the options and methods are exactly the same. The difference is that Radiobuttons are not independent. Of all the Radiobuttons whose values are reported and controlled by the same IntVar, exactly one of them must be on at all times.



There is in fact an exception to the “exactly one” rule. There is nothing the user can do to make any more or less than one option be selected, but the program can. This is the code that produced that display:

```
1 >>> iv1 = IntVar(value = 2)
2 >>> lbl = Label(win, text = "Select the size")
3 >>> lbl.grid(row = 0, column = 0, sticky = "w")
4 >>> cb1 = Radiobutton(win, variable = iv1,
5 ...                       text = "small", value = 1)
6 >>> cb1.grid(row = 1, column = 0, sticky = "w")
7 >>> cb2 = Radiobutton(win, variable = iv1,
8 ...                       text = "medium", value = 2)
9 >>> cb2.grid(row = 2, column = 0, sticky = "w")
10 >>> cb3 = Radiobutton(win, variable = iv1,
11 ...                       text = "large", value = 3)
12 >>> cb3.grid(row = 3, column = 0, sticky = "w")
```

The three Radiobuttons all use the same IntVar, `iv1`. That means that only one of them may be selected. Another group of Radiobuttons using a different IntVar will be independent from this group.

The value option says that value that the IntVar will have when this option is selected, and of course the other way round: if the IntVar’s value is changed the Radiobutton with the matching value becomes the selected one. This is how the program can disobey the “exactly one” rule. Change the IntVar so that it matches none of the options. The default value for an IntVar is zero, not explicitly setting it would have resulted in nothing being selected. That is a useful way to set things up

if you want to be sure that the user has made a positive selection and not just accidentally taken your default.

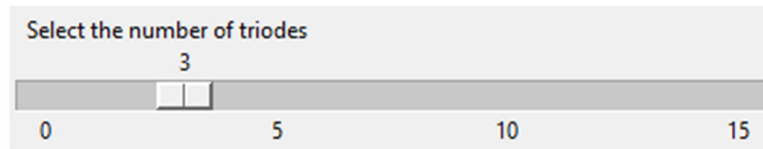
These options are exactly the same as for Checkbuttons: background or bg, foreground or fg, borderwidth or border or bd, text, textvariable, image, selectcolor, selectimage, compound, font, justify, cursor, padx, pady, command, height, width, relief, offrelief, overrelief, indicatoron, state, underline, wraplength, anchor, disabledforeground, activebackground, and activeforeground.

These four methods are the same as for Checkbuttons: select, deselect, flash, and invoke. There is no toggle.

And there isn't anything else.

## 50. Scale

A `Scale` allows the user to select a numeric value from a predetermined range by sliding a marker up and down or left and right along a marked background.



This is the code that produced that scale:

```
1 >>> iv1 = IntVar(value = 3)
2 >>> sc = Scale(win, label = "Select the number of triodes",
3 ...         variable = iv1,
4 ...         from_ = 0, to = 15,
5 ...         length = 400,
6 ...         orient = "horizontal",
7 ...         tickinterval = 5)
8 >>> sc.pack()
```

If you don't specify a label, there won't be one and the widget will be narrower. The number above the slider shows the value currently selected, if you don't want it to appear, set `showvalue = 0`. `variable` is as with Checkbuttons and Radiobuttons, it both controls and is controlled by the `Scale`. The variable may be an `IntVar`, a `DoubleVar`, or a `StringVar`. Using a `DoubleVar` means that non-integer values can be selected. `StringVar` is rather pointless, with it you can still only select numeric values, but that value will be presented as a string.

`from_` and `to` specify the minimum and maximum selectable values, the underline after `from` is because `from` is a Python reserved word. `length` is the length of the whole bar in pixels. `orient` may be "horizontal" or "vertical". `tickinterval` specifies which numbers will appear beneath the bar. The `from_` value will always appear, and after that numbers appear at the given interval. Set `tickinterval` to 0 if you don't want any numbers to appear at all.

These options are exactly the same as for Checkbuttons and Radiobuttons: background or bg, foreground or fg, borderwidth or border or bd, selectcolor, font, cursor, relief, state, and activebackground.

There are no options for providing a disabled... colour scheme, or for changes in relief, or for an anchor, or for a textvariable.

`command = callback`

Specifies a function that will be called every time the user changes the selection. This time it has one parameter, which will be the newly selected value. Strangely, it is always presented as a string.

`width = number`

The size, in pixels, of the narrow dimension of the slider and troughs to either side of it.

`resolution = number`

The only values that can be selected are equal to `from_` plus an integer multiple of `resolution`. If set to `-1` and `variable` is a `DoubleVar` no rounding is performed. On my Windows PC, `resolution` doesn't work properly when a `DoubleVar` is used. I can't see any pattern behind what actually happens.

`repeatdelay = milliseconds`

`repeatinterval = milliseconds`

The same as for a `Scrollbar`, used when the user holds the mouse button down in one of the troughs beside the slider.

`troughcolor = colour`

The colour of the troughs on either side of the slider.

`digits = int`

The behaviour of this option is not consistent, at least on my Windows PC. If I specify a value for `digits`, then the string passed to the callback function is always in the form of a float with `digits` minus two digits after the decimal point. It has no effect on what is stored in the controlling variable when it is a `StringVar`.

The `Scale`'s extra methods are:

`sc.get()`

`sc.set(newvalue)`

Retrieve or change the currently selected value.

`sc.coords()`

Returns `(x, y)` being the position of the centre of the slider, relative to the top left corner of the `Scale` widget.

`sc.coords(value)`

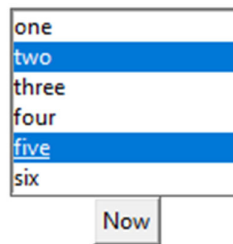
Returns (x, y) being the position that the centre of the slider would have, relative to the top left corner of the Scale widget, if it were set to this value.

`sc.identify(x, y)`

(x, y) are pixel coordinates relative to the top left corner of the Scale widget. The returned value will be "trough1", "slider", or "trough2" depending on what is at those coordinates, or the empty string if none of them are. The troughs are numbered the same was as for a Scrollbar.

## 51. Listbox

A Listbox displays a number of options as text in a box. The user selects any of them by clicking on them with the mouse.



This is the code and output produced after I first clicked on “five”, then on “two”, and finally on the “Now” button:

```
1 >>> def doit():
2 ...     cs = lb.curselection()
3 ...     print("selection is", cs)
4 ...     for i in cs:
5 ...         print("  includes", lb.get(i))
6
7 >>>
8 >>> sv = StringVar(value = "one two three four five six")
9 ... lb = Listbox(win, listvariable = sv,
10 ...             height = 6,
11 ...             selectmode = "multiple")
12 >>> lb.grid(row = 0, column = 0)
13 >>> bu = Button(win, text = "Now", command = doit)
14     bu.grid(row = 1, column = 0)
15
16     selection is (1, 4)
17         includes two
           includes five
```

The options that a Listbox will display are given to it as space separated words in the value of its controlling StringVar. You can change the choices at any time just by using the StringVar’s set method. If you want the choices to have spaces within them, prefix those spaces with \ characters. "one\ two\ three four five\ six"

specifies three choices that will occupy three lines in the `Listbox`. Those choices are “one two three”, “four”, and “five six”. Provide the `StringVar` through the `listvariable` option.

`height` specifies the number of lines the `Listbox` will show, the default is 10. If that is smaller than the number of choices, a `Scrollbar` will *not* be provided automatically, but the user can still browse through them all by using the up and down keyboard keys or the mouse. You can attach a vertical `Scrollbar` to a `Listbox` if you want to.

`selectmode` specifies what the user is allowed to choose. “single” and “browse” both mean you can only select a single item. “extended” means that you can select any number of neighbouring items (drag the mouse over them or use the keyboard up and down keys while holding shift down), and “multiple” means you can select any items you want (click once to select, click again to deselect). The difference between “single” and “browse” is very slight, and affects only the result of a mouse drag across multiple items. “single” means the item that you pressed the mouse button over is selected, “browse” means the item that you released the mouse button over is selected. The default is “browse”.

The two methods used in the `doit` function behave as the example illustrates. `lb.currselection()` returns a tuple of all the currently selected items’ indexes, empty if nothing is selected. `lb.get(i)` returns the string that is the  $i^{\text{th}}$  option, counting starts with 0 for the first option.

These additional options are the same as for just about everything: `background` or `bg`, `foreground` or `fg`, `borderwidth` or `border` or `bd`, `font`, `cursor`, `relief`, `state`, and `disabledforeground`.

`width` is the width of the box in characters.

`selectforeground` and `selectbackground` are the colours that will be used for the selected items.

`selectborderwidth`, default 0, puts an extra border around every choice, regardless of whether it is selected or not. Normally that border is the same as the background colour, but for a selected item, a raised border in shades of the background colour is shown.

`activestyle` allows for more changes in the appearance of selected items, the possible values are “none”, “underline”, and “dotbox”. This option does nothing on my Windows PC, so I can’t say what they are really supposed to look like.

Many of the `Listbox` methods refer to a particular one of the choosable strings. When I call a parameter “index”, it can be any one of an int that specifies a line number starting from 0, “active” to mean the line that was last selected, “end” for the last line in the `Listbox`, or a string of the form “@x,y” where x and y are



pixel coordinates relative to the `Listbox`'s top left corner. It refers to the line that is closest to that position.

`lb.itemconfig(index, option = colour, option = colour, ...)`

Sets a special option for a single line. Option may only be `selectbackground`, `selectforeground`, `background`, or `foreground`.

`lb.itemcget(index, option)`

Option must be a string, and may only specify one of the four options accepted by `itemconfig`. Returns the value of the given option for that line.

`lb.size()`

Returns the number of lines in the `Listbox`. Not just the visible ones, but all of them.

`lb.curselection()`

As in the example, a tuple of all the selected indexes.

`lb.selection_includes(index)`

Returns `1` if the given line is currently selected, `0` if it isn't..

`lb.selection_clear(index1, index2)`

De-select all lines in the given inclusive range. `index2` defaults to being equal to `index1`, but may also be `"end"`.

`lb.selection_set(index1, index2)`

The same as `selection_clear`, except that it selects the indicated lines. It does not replace the current selection, the given range of lines becomes selected along with those that already were. This can cause multiple non-adjacent items to be selected even when `selectmode` is `"single"`, `"extended"`, or `"browse"`.

`lb.see(index)`

Scrolls the options, if necessary, so that the indicated item is visible. Not necessarily at the top or bottom, just somewhere in sight.

`lb.get(index1, index2)`

Returns a tuple of all the option strings between the two indexes inclusive (not the usual Python way, but it is the usual `tkinter` way). `get(1, 3)` returns the second, third, and fourth options. `index2` is optional, it defaults to being equal to `index1`.

`lb.insert(index, string, string, string, ...)`

Adds new options just before the given position, but in this context, `"end"` means add it as a new last item. Unlike in the controlling `StringVar`, each option gets its own string, so do not use `\` before spaces.

`lb.delete(index1, index2)`

Removes all the choices between the two inclusive positions. `index2` is optional, if not provided, just one item is removed.

`lb.bbox(index)`

Returns the tuple (`leftx`, `topy`, `width`, `height`) being the bounding box for the indicated line, with `x` and `y` relative to the `Listbox`' top left corner. If the indicated line is not visible, it returns `None`.

`lb.nearest(y)`

Returns the index of the *visible* line that is nearest to the given `y` position, measured in pixels from the top of the `Listbox`. If the `Listbox` has no items in it at all, this returns `-1`.

`lb.selection_anchor(index)`

The documentation leaves me without the slightest idea of what this is supposed to do, and in every experiment I have been able to think of, it has no effect. Perhaps it's another exception for Windows.

`lb.index(index)`

Scrolls the options so that the indicated one is as close to the top of the box as possible. It has no effect on my Windows PC.

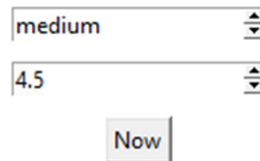
`lb.activate(index)`

Select that line. This has no effect on my Windows PC.

Scrollbars, both vertical and horizontal, may be added to a `Listbox` in the normal way, using the `xscrollcommand` and `yscrollcommand` options and the `xview`, `yview`, `scan_mark`, and `scan_dragto` methods.

## 52. Spinbox

A `Spinbox` is a restricted kind of `Listbox`. It can only ever display one option at a time, and only a single option can ever be selected. At the right edge of a `Spinbox` there are two tiny arrows, ▲ and ▼. Clicking on those changes what the selection is.



As well as clicking on the arrows, the user can click on the text itself and enter their own value by typing and using the backspace key and whatever. Unfortunately, there is no checking. The user is free to enter anything they like and it will be accepted. That can be fixed if you specify "readonly" as the state.

This is the code that produced this and the results after I clicked the top box' up arrow twice, the second box' up arrow nine times, clicked the now button, clicked the second box's down arrow three times, the finally clicked the now button again.

```

1 >>> def now():
2 ...     print(sb1.get(), "and", tv.get())
3
4 >>> tv = StringVar()
5 >>> sb1 = Spinbox(win, values = ("tiny", "small", "medium",
6 ...                             "large", "big", "enormous"))
7 >>> sb1.grid(row = 0, column = 0, pady = 5)
8 >>> sb2 = Spinbox(win, from_ = 0, to = 10, increment = 0.5,
9 ...               textvariable = tv)
10 >>> sb2.grid(row = 1, column = 0, pady = 5)
11 >>> bu = Button(win, text = "Now", command = now)
12 >>> bu.grid(row = 2, column = 0, pady = 5)
13
14
15     medium and 4.5
16     medium and 3.0

```

As you can see, there are two ways to specify the options: either a tuple of values (they can be just about anything that has a simple form when printed, and they don't have to be of the same type), or through the `from_`, `to`, and resolution options that we saw for a `Scale`.

There are also two ways of finding the currently selected option: the `Listbox` has its own `get` method, or you can provide a `StringVar` as the `textvariable` option and use its `get` method. Both ways will always deliver a string result.

By default, The `Listbox` first appears with its first option selected, but you can change that if you provide the `textvariable` option with a `StringVar` and set its initial value to the exact form that `get` would deliver for the option you want to appear. But pre-setting the `StringVar` only seems to work if the options were defined with `from_` and `to`, not with a tuple. There are a lot of methods that are supposed to let the program choose what is selected, but at least on my Windows PC, they have no effect. Except for the `invoke()` method, described below.

Other options include the usual familiar ones: `background` or `bg`, `foreground` or `fg`, `borderwidth` or `border` or `bd`, `selectcolor`, `disabledbackground`, `disabledforeground`, `font`, `justify`, `cursor`, `relief`, `wrap`, `selectbackground`, `selectforeground`, `selectborderwidth`, and `activebackground`.

As well as those, `insertbackground` and `insertborderwidth` control the appearance of the text input cursor when the user chooses to type their own value.

`command` is a parameterless callback function, activated only when the user clicks on one of the arrows and the selection changes as a result.

`format` should be set to one of the allowed `%` formats, e.g. `"%8.1f"`, and gives the format for converting options to strings. It only works if you used `from_` and `to`, not values.

`state` can take on the two usual values `"normal"` and `"disabled"`, but it may also be `"readonly"` which isn't quite the right term. It means that the selection can still be changed by clicking on the arrows, but the user can't type their own value any more. `readonlybackground` sets the colour to be used when the state is `"readonly"`.

`width` is, as is normal for text based widgets, gives the width measured in characters.

`insertbackground`, `insertwidth`, `insertontime`, and `insertofftime` are as they were for an `Entry`. There is supposed to be an `insertborderwidth` too, but it has no effect on my Windows PC.

`repeatdelay` and `repeatinterval` are the same as they are for a `Scrollbar`.

`buttonbackground` specifies the background colour for the arrows, `buttoncursor` sets the cursor that will appear when the mouse is above the buttons, allowable values are as described under canvas options. `buttonuprelief` and `buttondownrelief` set the relief for the arrows, but the arrows are far too small for that to have any useful effect.

These are the methods for a `Spinbox`, remember that all the universal methods are also available.

There are a few methods whose names begin with `selection` which, according to the documentation, are supposed to change which option is selected. On my Windows PC they have no effect, and the documentation doesn't say exactly what their effect is supposed to be.

`sb.bbox(index)` is supposed to show the bounding box for the given option, just like for a `Listbox`, but it is rather silly because in a `Spinbox` there is only one place that an option can appear, so it always returns the same four-tuple. `sb.index` and `sb.insert` have some strange effect, but nothing that seems at all useful.

`sb.invoke("buttonup" or "buttondown")` has the same effect as the user would cause by clicking on the given arrow.

`sb.delete(index1, index2)` has no effect, at least on my Windows PC.

`sb.icursor(position)` sets the position of the keyboard input cursor if there is one, and is measured in characters, 0 meaning before the first.

`sb.identify(x, y)` returns "entry", "buttonup", or "buttondown" depending on which of those things is at the position (x, y) measured in pixels from the widget's top left corner, or the empty string if nothing is.

If the `Spinbox` is likely to have some very long options, you can attach a horizontal `Scrollbar` to it in the normal way.

## 53. Frames - windows within windows

A `Frame` is a lot like a window. It can have a lot of widgets packed, placed, or gridded inside it. But it is a widget itself, and can be packed, placed, or gridded into something else. This can simplify the design of a complex gui and remove the need for so many `rowspans` and `columnspans`.

`Frames` have many of the familiar options, plus the universal options and methods of course: `background` or `bg`, `borderwidth` or `border` or `bd`, `padx`, `pady`, `relief`, and `cursor`.

They also have `width` and `height` options which specify the desired size in pixels. But these two are usually ignored, container widgets are supposed to adopt a size that fits their contents perfectly. If you want a `Frame` to take the size you specify, you must also call its `grid_propagate` method with 0 as its parameter.



```
1 >>> win = Tk()
2 >>> win.configure(background = "white")
3 >>> win.configure(padx = 15, pady = 15)
4
5 >>> one = Label(win, text = "Label One")
6 >>> one.grid(row = 0, column = 0)
7 >>> two = Label(win, text = "Label Two")
8 >>> two.grid(row = 0, column = 1)
9 >>> three = Label(win, text = "Label Three")
10 >>> three.grid(row = 1, column = 1)
11 >>> fr = Frame(win, padx = 20, pady = 20)
12 >>> fr.grid(row = 1, column = 0)
13
14 >>> four = Label(fr, text = "Label Four", background = "yellow")
15 >>> four.grid(row = 0, column = 0)
16 >>> five = Label(fr, text = "Label Five", background = "white")
```

```

17 >>> five.grid(row = 0, column = 1)
18 >>> six = Label(fr, text = "Label Six", padx = 20, pady = 20)
19 >>> six.grid(row = 1, column = 0)
20 >>> seven = Label(fr, text = "Label Seven", foreground = "red")
21 >>> seven.grid(row = 1, column = 1)

```

It seems to be impossible use Scrollbars to control a Frame. That is a surprising thing to be absent.

If widgets were inserted into a Frame or other container using grid, the container and the widgets that it contains have some useful methods. In these methods you must be careful about rows and columns because they are used completely inconsistently. Sometimes rows comes first, and sometimes columns comes first.

There are the methods to use on the container:

```
fr.grid_size()
```

Returns a tuple (c, r) where c is the number of columns in the grid, and r is the number of rows.

```
fr.grid_slaves()
```

Returns a list of all the widget objects that are in the grid.

```
fr.grid_location(x, y)
```

x and y are pixel coordinates measured from the top left corner of the container. Returns a tuple (column, row) telling you which grid position x and y are within.

```
fr.grid_propagate(0 or 1), 1 is the default setting.
```

Normally the width and height you specify when creating a Frame are ignored, and it is sized to fit its contents. fr.grid\_propagate(0) changes this, the Frame will have the size you told it to have.

```
fr.grid_bbox()
```

Returns a four-tuple (x, y, width, height) giving the bounding box of the whole grid. All four are measured in pixels, with x and y being relative to the container's top left corner.

```
fr.grid_bbox(column, row)
```

Returns the bounding box for the grid cell at the given position. That is not the same as the bounding box for the widget at that position. Grid sizes stretch to accommodate their largest widget.

```
fr.grid_bbox(column, row, col2, row2)
```

Returns the bounding box for the rectangular region of the grid that has cell (row, column) at its top left and cell (col2, row2) at its bottom right.

There are the methods to use on the contained widgets:

```
w.grid_remove()
```

Takes the widget out of the grid, it disappears from the screen. But you can put it back again with a parameterless w.grid() later on.

```
w.grid_forget()
```

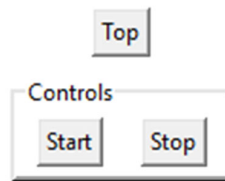
This is a more violent version of `grid_remove`. The widget disappears from the grid as before, but its position and other things given as parameters when it was originally gridded are not recorded. You can put it back in the grid later on, but you need to give `w.grid(...)` all the parameters that you would use to place a new widget.

```
w.grid_info()
```

Returns a dictionary of all the aspects of the widget that were or could have been provided as parameters to `grid`. They are `row`, `column`, `rowspan`, `columnspan`, `padx`, `pady`, `ipadx`, `ipady`, and `sticky`. This is *not* a reference to some real object in `tkinter`, changing one of the dictionary's values will have no effect on the widget itself.

## 54. LabelFrame

`LabelFrame` would seem to be a subclass of `Frame`. It isn't, but still has everything that a `Frame` has, so those things will not be repeated here. The difference with a `LabelFrame` is that it draws a box around its contents with a gap in it. That gap is usually filled with some descriptive text, but you can put anything you want in there.



```
1 >>> top = Button(win, text = "Top")
2 >>> top.pack(pady = 5)
3 >>> lfr = LabelFrame(win, text = "Controls",
4 ...           border = 4,
5 ...           relief = "raised",
6 ...           background = "white")
7 >>> lfr.pack(pady = 5)
8 >>> b1 = Button(lfr, text = "Start")
9 >>> b1.grid(row = 0, column = 0, padx = 10, pady = 5)
10 >>> b2 = Button(lfr, text = "Stop")
11 >>> b2.grid(row = 0, column = 1, padx = 10, pady = 5)
```

Obviously, text is what appears in the gap, border is the width of the edges of the box (default 1), relief is how the box is drawn (default "groove"), and background is what it always is. Additional options that `Frames` don't have are:

`foreground` or `fg`

The colour for the text, but not the border.

`labelwidget`

Any widget at all, it will appear in the gap instead of text.

labelanchor

A string that says where in the surrounding box the gap and its `text` or `labelwidget` will be. There are twelve possible values.

"nw", "n", and "ne" mean they will be at left, centre, or right of the top edge,

"sw", "s", and "se" are the left, centre, or right of the bottom edge,

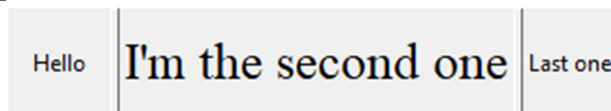
"wn", "w", and "ws" are the top, centre, or bottom of the left edge, and

"en", "e", and "es" are the top, centre, or bottom of the right edge.

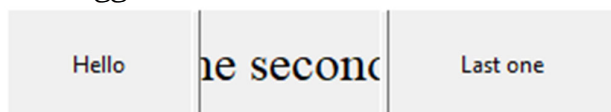
## 55. PanedWindows

`PanedWindows` are another kind of container. In this case the widgets that it contains can only be lined up horizontally or vertically, no `grid`. That isn't a disadvantage, it is necessary in order for it to work. The contained widgets, which would normally be fairly big things like `Canvasses`, `Texts`, or `Frames` are shown with a separator between them. The user can drag those separators with the mouse and change the sizes of the components. If the separator is dragged to the left, the widget to the left of it shrinks, and the widget to the right grows to take up the space. The separators are supposed to be called sashes. On my Windows PC, the default is for the sashes to be completely invisible. I would call that a strange choice.

This is the initial appearance:



and this is after I have dragged both sashes inwards.



```
1 >>> pw = PanedWindow(win, orient = "horizontal",
2 ...                 sashwidth = 5,
3 ...                 sashrelief = "raised")
4 >>> pw.pack()
5
6 >>> one = Label(pw, text = "Hello")
7 >>> pw.add(one, padx = 10)
8
9 >>> two = Label(pw, text = "I'm the second one",
10 ...           font = ("times new roman", 20))
11 >>> pw.add(two, pady = 10)
12
13 >>> three = Label(pw, text = "Last one")
14 >>> pw.add(three)
```

If `orient` is "vertical", then naturally the contents will be stacked vertically and the sashes between them will be horizontal. The `sashwidth` and `sashrelief` were necessary to make them reasonably visible, and even now I don't think they stand out at all well. The `background` or `bg` option sets the background colour



which is also the sash colour, so that could make them stand out better, but then you would have to make sure that the background never shows naturally anywhere else, and you would have to set `bordersize` or `border` or `bd` to 0, and you couldn't use `padx` or `pady`. It is not possible for the background and the sashes to have different colours.

The `add` method is the only way to add a widget as a new pane, do not use `pack`, `place`, or `grid`.

Additional options for the `PanedWindow` constructor include the usual `cursor`, `width`, and `height`, plus:

`showhandle` = `True` or `False`, `False` is the default.

If set to `true`, this makes little square boxes, "handles", appear on the sashes, giving the user a larger target area for the mouse.

`handlesize` = `number`, 8 is the default.

Specifies the width and height of the handles.

`handlepad` = `distance`, 8 pixels is the default.

This is the distance between the top or left edge of sash and the top or left edge of the handle.

`opaqueresize` = `True` or `False`, `True` is the default.

Normally the display is updated in real time when the user drags a sash, continually showing the panes in their new size. Turning `opaqueresize` to `False` stops this, the display isn't changed until the user releases the mouse.

`handlepad` = `size`

This adds some padding between the sashes and the contents of the panes. Unfortunately the background colour shows through in this extra space.

The `add` method has some options of its own:

`width` = `size`

`height` = `size`

The size you want the added widget to have.

`minsize` = `size`

The smallest width or height the widget can be shrunk to when the user drags a sash on either side of it.

`padx` = `size`

`pady` = `size`

Extra spacing around the added widget, through which the background colour will show.

`sticky`

Exactly the same as for `grid`.

`before = otherwidget`

`after = otherwidget`

Otherwidget must be a widget that has already been added. Instead of being added after all the existing widgets as usual, this one will be pushed in immediately before or after the otherwidget.

Other methods for a `PanedWindow` are:

`pw.remove(widget)`

`pw.forget(widget)`

For a `PanedWindow`, these two methods are identical, they both make a pane disappear, and it can always be readded again later. Unlike with a `Frame`, `remove` does not record the original position of the widget.

`pw.panes()`

Returns a list of all the contained widgets in the same order as they currently appear on the screen.

`pw.sash_coord(which)`

Returns the position of a particular sash as an `(x, y)` tuple. `x` and `y` are measured in pixels from the top or left of the `PanedWindow`. Only one of `x` and `y` will be correct, which one depends on `orient`. Which is an int, 0 means the leftmost or topmost sash, and of course they are counted in ones from there.

`pw.sash_place(which, x, y)`

The parameters are as for `sash_coord`. It moves a sash to the given position. Depending on the `orient`, either `x` or `y` will be ignored.

`pw.paneconfig(widget, option = value, option = value, ...)`

Changes the options that were given when the widget was added.

`pw.paneconfig(widget, option)`

Option must be a string, equal to one of the option names that are allowed for the `add` method. Returns the value of that option for the given widget.

`pw.identify(x, y)`

`x` and `y` are pixel coordinates measured from the top left corner of the `PanedWindow`. Returns what is at that position, either `(which, "sash")` or `(which, "handle")` if one of those is true, where `which` is a sash number. If the coordinates are outside the `PanedWindow`, or over one of the added widgets, it returns an empty string.

## 56. The Tk window

Here are all the methods for the root window created by the call to `Tk()`:

`win.destroy()`

Permanently removes the window from the screen and destroys its contents.

`win.protocol("WM_DELETE_WINDOW", callback)`

Prevents the user from closing the window. That's what the documentation says, but I recognise `WM_DELETE_WINDOW` as something specific to the Windows operating system. Maybe they decided to use that name for all operating systems, but don't rely on that. If the user clicks on the window's close button, instead of closing the window, the parameterless callback function will be called. It can do any saving and closing that is necessary, and they use `win.destroy` if desired.

`win.title(string)`

Sets the title that appears at the top of the window. If you don't provide the parameter, it returns the current title as a string.

`win.iconbitmap(filename)`

Changes the icon that appears in the top left corner of the window, and also represents the program when its window has been minimised or iconified. The filename is a string like `"d:\\pictures\\logo.ico"`, and the file *must* be in the `.ico` format. Not many image editing applications support that format, even Photoshop doesn't. But there are numerous free on-line services that will do it in an instant.

What should the dimensions of the image be? Unfortunately that depends on your operating system. If you just want a tiny little icon at the top of a window, they seem to be  $16 \times 16$ . The representation of a minimised window seems to be  $32 \times 32$ , and there are other possibilities. But most systems, Windows included, will scale an icon if the file is not of the size that it wants. Scaling up from a small image is never good, so for complete generality,  $256 \times 256$  should cover everything.

`win.geometry(string)`

Changes the size and position of the window. The string should be of this format `"640x480"` if you just want to change the size. That would be 640 pixels wide and 480 tall. Use this format `"640x480+200+100"` to change the position too. That would place the window 200 pixels in from the left of the screen and 100 down from the top. If you do not provide a parameter, it will return the correct string to represent the window's current state.

`win.attributes("-fullscreen", 1)`

Expands the window to fill the whole screen. The dash in the name is essential. Using 0 instead of 1 changes it back to its original size.

`win.wininfo_screenwidth()` and

`win.wininfo_screenheight()`

Return the ints that you would expect from their names. They can be helpful in deciding the correct geometry for your window.

`win.resizable(w, h)`

Allow you to prevent the user from changing the size of the window. `w` and `h` should both be either `True` or `False`. `True` is the default state for a window. `w` being `False` means that the width can not be changed, `h` refers to the height.

`win.minsize(w, h)` and

`win.maxsize(w, h)`

Are less restrictive than `win.resizable`. They specify the minimum size that the user may shrink the window to, and the maximum size that the user can expand it to.

`win.iconify()` or

`win.deiconify()`

Make the window disappear to just be represented by a little icon, usually at the bottom of the screen, or be restored from that state.

`win.withdraw()` and

`win.state()`

Is like `win.iconify` but the window completely disappears without even leaving an icon to represent it, so the user has nothing to click on to restore it. Also use `win.deiconify()` to restore it, there is no `dewithdraw`. `state` returns one of `"normal"`, `"iconic"`, or `"withdrawn"`.

`win.attributes("-topmost", 1)`

Brings the window to the front, above all other windows, and keeps it there. The user can not make any other window cover it. This is used when there is some unusual condition or emergency that the user must deal with. Using `0` instead of `1` doesn't push the window back down under others, it just restores it to its normal compliant state, so the user can bring others above it.

`win.attributes("-disabled", 1)`

Makes the window totally inert, nothing the user does has any effect on it. Using `0` instead of `1` returns it to normal. `win.state` is unaware of windows being disabled, and continues to report them as `"normal"`. Some system change the appearance of disabled windows, but Windows leaves them looking exactly as they did.

`win.attributes("-alpha", float)`

Makes the window transparent, float should be between `0` and `1`. `0` makes the window totally invisible, `1` makes it completely opaque. `0.5` or other intermediate values make it partially transparent, you can still see it, but you can also see other things that are on the screen behind it.

`win.overrideRedirect(1)`

Is a strange name for what it does. It takes away the window's title bar and any curved or raised edges it might have, leaving just the rectangle for its contents. In this state, the user can not move it or close it or minimise it. Using `0` instead of `1` puts everything back to normal.

## 57. Toplevel: an independent window

A `Toplevel` is a separate totally independent window just like the one created by `Tk()`, it does not get inserted into any container. Like any other container, widgets may be packed, placed, or gridded into it. The window can have a title, a menu bar, and everything you would expect of a window. Oddly, there is no `title` option, you have to use the `title` method after creating the window. A `Toplevel` is counted as a widget itself, even though it can't be put into a container. If you create a widget before creating a `Tk()` window, an unusable `Tk()` will be created automatically. That means that you should not try to use a `Toplevel` as your program's main window, you'll end up with a second grey window just looking silly. `Toplevels` are used as secondary windows, if you need to create a temporary dialogue or something like that.

```
win2 = Toplevel(option = value, option = value, ...)
```

The options are the usual `background` or `bg`, `borderwidth` or `border` or `bd`, `width`, `height`, `relief`, `padx`, `pady`, and `cursor`. `borderwidth` does not make any border appear, it just has the effect of setting both `padx` and `pady`, so that added widgets are kept away from the edges of the window. On my Windows PC, `relief` has no effect.

There is an additional option, `menu`, which is used to add a standard menu bar. Menus are covered in a later subsection.

`Toplevel` has all of these methods exactly as they are for a `Tk()` window: `destroy`, `title`, `protocol`, `iconbitmap`, `attributes` (with the same options), `resizable`, `minsize`, `maxsize`, `iconify`, `deiconify`, `withdraw`, `state`, and `overrideredirect`.

## 58. Ttk widgets

`tkinter.ttk` provides more widgets. Most of them are almost exact copies of the familiar ones: `Label`, `Button`, `Entry`, `Scrollbar`, `Checkbutton`, `Radiobutton`, `Scale`, `Spinbox`, `Frame`, `LabelFrame`, and `PanedWindow`. It also provides six new kinds of widgets that will be covered in the next few subsections.

`ttk` widgets are called *styled* widgets, which means that you can define a lot of styles (collections of common options like `background` and `foreground`) and apply an entire style to a `ttk` widget as a single option.

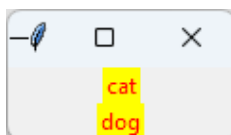
Every `ttk` widget has its own predefined default style, and the name of that style (actually called its class name) is nearly always a capital `T` followed by the name of the type of widget, such as `"TButton"`. There are five exceptions, some of them just plain foolish. The three kinds of widget that can have an orientation are like `"Vertical.TScrollbar"`: their natural names preceded by their orientation

(with a capital first letter) and a dot. A `PanedWindow` loses the capitalisation of its `W`: `"TPanedwindow"`, and the new widget `Treeview` gets no additional `T`: `"Treeview"`.

In case of future changes or additions, there is always an easy way to find the correct class name. Just create a minimal widget `widg` of that type, and get it to tell you by asking `widg.winfo_class()`. The result will be a string, like `"TLabel"`.

When you create your own style, it must be a variation on an existing style, and its name absolutely must be an arbitrary non-empty string, followed by a dot, followed by the name of an existing style, such as `"redandyellow.TLabel"`.

```
1 >>> from tkinter import *
2 ... from tkinter import ttk
3 ...
4 >>> win = Tk()
5
6 >>> def pressed():
7 >>>     print("Pressed!")
8
9 >>> redyel = ttk.Style()
10 ... redyel.configure("redandyellow.TLabel",
11 >>>                     foreground = "red",
12 >>>                     background = "yellow")
13 >>> lab = ttk.Label(win, text = "cat",
14 >>>                  style = "redandyellow.TLabel")
15 >>> lab.pack()
16 >>> but = ttk.Button(win, text = "dog",
17 >>>                  style = "redandyellow.TLabel",
18 >>>                  command = pressed)
19 >>> but.pack()
```



The example shows two surprising things. You can apply a style made for one kind of widget (here a `Label`) to a totally different kind of widget (here a `Button`). The result is a `Button` that looks exactly like a `Label`, not even any visual change when it is clicked. But it still functions as a `button`, clicking it does call its callback function.

There is another surprising and difficult to handle inconsistency. If you mistype the name of the style when creating the widget (I typed a little `l` instead of a capital `L`) you get a proper exception and know about it. You can't use a style that doesn't exist. But if you make the same mistake when creating the style, but don't make that mistake when using it, you are still trying to use a style that doesn't exist, but no exception is raised, it just silently fails.

There is a quite complicated and poorly documented hierarchy of things going on here. First of all, there is a *theme*, this is intended as a way of giving all of your widgets the same, or at least a consistent look. An example is that Macintosh gui elements usually have a rounded shape, but Windows makes most things rectangular. That could be called a theme.

Naturally, the appearance can't be exactly the same for every kind of gui object, otherwise we would have buttons that are indistinguishable from labels, like in the example above, and nobody would know what is going on. Each kind of object needs its own detailed description to specify how the theme is applied to it. That is what a *style* is. A theme is a collection of styles.

Gui objects tend to consist of a number of parts all put together. For example, a `Button` may have an outer border, inside that border there may be but usually isn't a focus ring which can change its appearance when the `Button` has focus. Inside that there may be some padding, and inside that there is the text itself, called the label. These are all called *elements*. Each widget has a collection of elements which may have their own options. The elements have names, a `Button`'s padding is called "`Button.padding`". In the case of a `Button`, they are all inside each other, so the label is considered to be a child of the padding, which is a child of the focus, which is a child of the border. It isn't always that way. In a `Scrollbar`, the two arrows and what they are now calling the thumb instead of the slider are considered to be children of the trough, and the grip, which I assume is the four little lines visible in the "clam" example below, is a child of the thumb.

Fortunately you don't have to pay any attention to elements unless you are going to create your own style from scratch. For all the options that elements may share, each style only records one value. Changing the background of the label also changes the background of the padding. If two elements are allowed to have different values, the options will have different names. Just as in a `Scrollbar`, where we have both `background` and `troughcolor` options.

To find out what themes are available, you need to create a `Style` object and use its `theme_name` method, it returns a tuple of strings. To find out what the current theme is, call its `theme_use` method. To change to a different theme, use its `theme_use` method again, but with the theme's name as a parameter.

```
1 >>> s = ttk.Style()
2 >>> s.theme_names()
3      ('winnative', 'clam', 'alt', 'default', 'classic', 'vista',
4       'xpnative')
5 >>> s.theme_use()
6      'vista'
7 >>> s.theme_use("winnative")
```

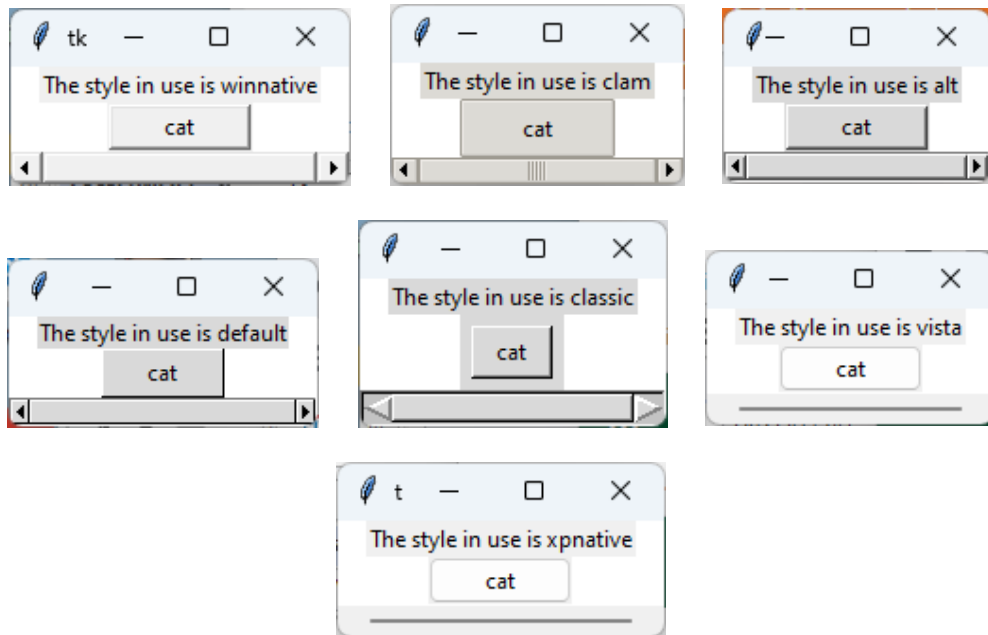
As you see, my Windows PC reports seven different themes, but that is quite deceptive, as they are all very similar. I made this little function to try them all out.

```

1 >>> def tryout(name):
2 ...     win = Tk()
3 ...     win.configure(background = "white")
4 ...     ttk.Style().theme_use(name)
5 ...     lab = ttk.Label(win, text = "The style in use is " + name)
6 ...     lab.pack(padx = 15)
7 ...     but = ttk.Button(win, text = "cat")
8 ...     but.pack()
9 ...     scr = ttk.Scrollbar(win, orient = "horizontal")
10 ...     scr.pack(expand = True, fill = "both")
11
12 >>> tryout("winnative")

```

And here are all the results:



If you know the name of a theme, or just want to deal with the default theme, you can extract a *layout* for any style. A layout lists all of the style's elements, their parent-child relationships, and supposedly all of their options. Layouts have very annoying formats, full of nested lists, tuples, and dictionaries. Rather than work through every detail, which would not be of any use because there is no other context where this comes into play, I'll provide a little program that just extracts all the element names. Every time a tuple is encountered in a layout, its first item will be an element name:

```

1 >>> def show_elements(theme_name, style_name):
2 ...     st = ttk.Style()
3 ...     st.theme_use(theme_name)
4 ...     lay = st.layout(style_name)
5 ...     explore_elements(st, lay)
6
7 >>> def explore_elements(st, item):
8 ...     if type(item) == tuple:
9 ...         print(item[0])
10 ...         explore_elements(st, item[1])

```



```

11 ...     elif type(item) == list:
12 ...         for i in item:
13 ...             explore_elements(st, i)
14 ...     elif type(item) == dict:
15 ...         for k in item:
16 ...             explore_elements(st, item[k])
17
18 >>> show_elements("classic", "TButton")
19     Button.highlight
20     Button.border
21     Button.padding
22     Button.label

```

And we can get a bit more out of it. When a tuple is encountered, after the element name, the next item will be a dictionary of all of that item's pre-set options, so a slightly more complex search is needed. Additionally, if you have a Style and an element name, the Style's `element_options` method returns a tuple of all the other permitted option names:

```

1 >>> def show_one_element(theme_name, style_name, element_name):
2 ...     st = ttk.Style()
3 ...     st.theme_use(theme_name)
4 ...     lay = st.layout(style_name)
5 ...     explore_one_element(st, lay, element_name)
6
7 >>> def explore_one_element(st, item, element_name):
8 ...     if type(item) == tuple:
9 ...         if item[0] == element_name:
10 ...             for opt in sorted(item[1]):
11 ...                 if opt != "children":
12 ...                     print(opt, ": ", item[1][opt], sep = "")
13 ...                 opts = st.element_options(element_name)
14 ...                 for opt in sorted(opts):
15 ...                     print(opt, "is also allowed")
16 ...         else:
17 ...             for opt in item[1]:
18 ...                 explore_one_element(st, item[1][opt], element_name)
19 ...     elif type(item) == list:
20 ...         for i in item:
21 ...             explore_one_element(st, i, element_name)
22 ...     elif type(item) == dict:
23 ...         for k in item:
24 ...             explore_one_element(st, item[k], element_name)
25
26 >>> show_one_element("classic", "TButton", "Button.label")
27     sticky: nswe
28     anchor is also allowed
29     background is also allowed
30     compound is also allowed
31     embossed is also allowed
32     font is also allowed
33     foreground is also allowed
34     image is also allowed
35     justify is also allowed
36     space is also allowed
37     stipple is also allowed

```

```

38     text is also allowed
39     underline is also allowed
40     width is also allowed
41     wraplength is also allowed

```

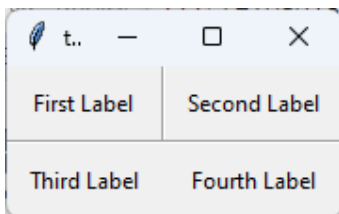
I won't waste space by showing all the options for the other three elements. One of them has none at all, the others all have `sticky`, and one also has `border`. But there is obviously something missing. Looking at the pictures, the `vista` theme gives Buttons rounded corners and `classic` makes them sharp right angles. Where is the option for that? There just isn't one. Themes and styles have hidden things that we just can't do anything about. Themes and styles are actually programmed in C. Nobody has produced a Python way of doing it. Well, maybe there is some third-party software somewhere, but I haven't seen it.

The `Separator` is the most trivial widget it is possible to imagine, it just appears as a straight line between other widgets. The only options it has are `orient` (as for a `Scrollbar`) and `style`.

```

1 >>> def lab(text, row, col):
2 ...     w = ttk.Label(win, text = text)
3 ...     w.grid(row = row, column = col, padx = 10, pady = 10)
4
5 >>> lab("First Label", 0, 0)
6 >>> sa = ttk.Separator(win, orient = "vertical")
7 >>> sa.grid(row = 0, column = 1, sticky = "ns")
8 >>> lab("Second Label", 0, 2)
9 >>> sb = ttk.Separator(win, orient = "horizontal")
10 >>> sb.grid(row = 1, column = 0, columnspan = 3, sticky = "ew")
11 >>> lab("Third Label", 2, 0)
12 >>> lab("Fourth Label", 2, 2)

```



## 59. `ttk.Progressbar`

We all know what a progress bar is, a coloured line that grows across the screen to let you know how far through a task the program is. They are usually horizontal, but they can be vertical if you want. It is controlled by an internal counter which starts at zero, you specify its maximum as an option (the default is 100), and tell the counter when to move and by how much using its `step` method. The `length` option specifies how long the bar should be in pixels, it is still called `length` even if the `Progressbar` is vertical.

```

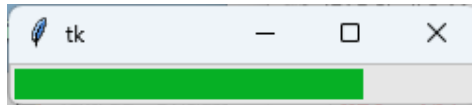
1 >>> pb = ttk.Progressbar(win, orient = "horizontal",
2 ...                       maximum = 50, length = 250)
3 >>> pb.pack()

```

```

4
5 >>> count = 0
6
7 >>> def move():
8 ...     global count
9 ...     pb.step(1)
10 ...     count += 1
11 ...     if count < 49:
12 ...         pb.after(100, move)
13
14 >>> move()

```



There are two mysteriously wrong things to keep in mind. One is that there doesn't seem to be any way to control the bar's colour or width. The other is that `maximum` does not behave very sensibly: the largest value the bar can reach is `maximum - 1`, at which point there is still visible space to the right. If the counter ever reaches `maximum`, the bar goes back to zero and disappears.

You may also be surprised by how complicated the controlling mechanism was. Why not just say

```

5 >>> for i in range(49):
6 ...     time.sleep(100)
7 ...     pb.step(1)

```

`tkinter` doesn't update the display immediately after every operation, it waits until nothing is going on, and a timed sleep somehow counts as something going on. With the loop version, the `Tk()` window doesn't even appear.

A widget's `after` method provides a period of time, in milliseconds, and a callback function, and it makes sure the function is called after the period of time has expired. The program would have been smaller if I had written a loop where `i` increases from 100 to 490 in steps of 100 and called `pb.after` with `i` as its first parameter each time round. That would have set up 49 callback requests all at once. But that approach is not scalable. Instead, the `move` function does its little bit of work, then requests another call to itself 100mS in the future.

A `Progressbar` has the usual `cursor` and `style` options, and one called `phase` which has no clear use, plus two more.

`value = number`

Provides a different starting value for the internal counter if you don't want it to start from zero.

`variable = IntVar or DoubleVar`

The `variable` provided replaces the internal counter. Just set its value and the bar moves. You can even make it move backwards.

mode = "determinate" (the default) or "indeterminate"

If you have no way of knowing how far through a task you are, make the `Progressbar` indeterminate. Then it will just move backwards and forwards until you tell it to stop. It is controlled by the `start` and `stop` methods.

`pb.start(t)`

Only for indeterminate `Progressbars`. This starts the bar's back and forth movement. Every `t` milliseconds it moves by one step. After reaching the maximum or minimum it just changes direction.

`pb.stop()`

Only for indeterminate `Progressbars`. This stops the bar from moving, call it when the task is complete. To avoid confusion, after calling `stop`, you should probably either set the bar to its maximum value or make it disappear altogether.

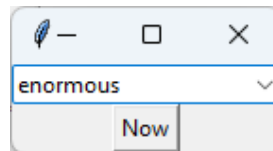
`pb.step(increment)`

Already seen, the increment is optional and defaults to 1.

## 60. `ttk.Combobox`

A `Combobox` is an extension of a `Spinbox` or `Listbox`. Both of those present the user with a list of possibilities to choose from. A `Combobox` also allows the user to type their own choice if it isn't among those offered. The fact is that a `Spinbox` already has that functionality, so we're not about to see any great advance.

```
1 >>> win = Tk()
2
3 >>> def doit():
4 ...     print("You selected \"", cb.get(), "\", sep = "")
5
6 >>> cb = ttk.Combobox(win, values = ("tiny", "small", "medium",
7 ...                                 "large", "big"))
8 ...     cb.pack()
9 ...     bu = Button(win, text = "Now", command = doit)
10 ...     bu.pack()
11
12     You selected "medium"
13     You selected "enormous"
```



The box initially appears completely blank, nothing is selected. You select something by clicking on the downward vee at the right of the box, a menu of the possibilities appears. I selected small, then selected medium, then pressed the button, then typed enormous into the box, then pressed the button again.

The options include the usual `justify`, `cursor`, and `style`, plus `validate` and `validatecommand` exactly as they are for an `Entry`. Unlike with the `Spinbox`, there is no `readonly` option.

The `background` option exists but is ignored, no exceptions raised. Trying to set `order` or `relief` does cause an `Exception`. `background`, `border`, and `relief` are just ignored if they are part of the selected style. A horizontal `Scrollbar` may be attached in the usual way.

`textvariable = StringVar`

The `StringVar` is bound to whatever string is selected, so using its `get` method is an alternative way of seeing what was chosen. Setting the `StringVar` changes the selection, giving it an initial value means that the `Combobox` will not start out blank with nothing selected.

`state = "normal" or "readonly" or "disabled"`

"disabled" prevents the user from typing their own choice.

`height = number`

This is the number of rows that the drop-down menu will have, except that the menu never has more lines than there are choices. The default is 20.

`width = number`

The width that the box will have, measured in characters.

`postcommand = function`

The callback function will be called every time the little vee at the right is pressed.

Other methods for a `PanedWindow` are:

`cb.current()`

`cb.current(position)`

The first form returns the index in the `values` option of the currently selected item, or `-1` if the selection was typed by the user and isn't in `values`. The second form returns the string at the given position in `values`.

`cb.set(string)`

Selects that string as though the user had typed it into the box.

`cb.bind('<<ComboboxSelected>>', callback)`

The callback function will be called every time the selection is changed by clicking on a different choice. The user typing their own value does not trigger this callback. This takes away the need for a button for the user to click when they have made their choice. The callback function will receive a pointless parameter of type `VirtualEvent` which carries no information at all.

And remember that widgets have configure methods. Any time you want to completely change the available choices, just say something like

```
cb.configure(values = ("big", "fat", "hairy"))
```

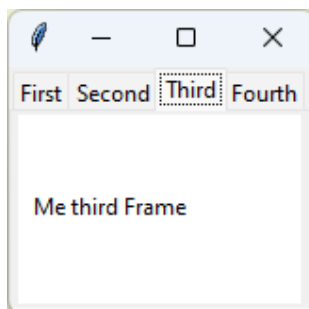
That will not change the current selection, but the available choices are completely replaced. If you just want to add a few options, leaving the existing ones intact:

```
cb.configure(values = cb.cget("values") + ("ugly", ))
```

## 61. ttk.Notebook

Notebooks allow a number of different versions of a widget, usually a Frame, to occupy the same position on the screen, with only one of them visible at any one time. There are a series of tabs, like with web browsers, that allow the user to select which version is visible.

```
1 >>> nb = ttk.Notebook(win)
2 >>> nb.pack()
3
4 >>> def make_frame(name, words):
5 ...     fr = Frame(nb)
6 ...     ca = Canvas(fr, width = 150, height = 100, background = "white")
7 ...     ca.pack()
8 ...     ca.create_text(10, 50, text = words, anchor = "w")
9 ...     nb.add(fr, text = name)
10
11 >>> make_frame("First", "I'm the first Frame")
12 >>> make_frame("Second", "I am the second Frame")
13 >>> make_frame("Third", "Me third Frame")
14 >>> make_frame("Fourth", "Fourth is my position")
```



All I did was click on the tab labelled “third” to get here. The Frames can contain all sorts of components, just like they usually do, and they don't even need to be Frames, any widget can be added to a Notebook.

A Notebook has cursor and style options, padding = dimension is added outside on all four sides, width and height are in pixels.

Many of the Notebook methods have to specify a particular tab. To do this, you can provide an int, being the tab's position, 0 for the first one. Or you can use the actual widget object that the tab contains, or you can give the string "current" to mean the one that is selected now, or if somehow you know the pixel coordinates

of some point inside the tab relative to the Notebook's top left corner you could provide the string "@120,10" for 120 left and 10 down. When a parameter is given as "tab" this is what it means.

The Notebook methods are:

`nb.add(widget, option = value, option = value, ...)`

Creates a new tab. The options are:

`text`, `image`, and `compound`:

as usual. The label on the tab can be a `PhotoImage` instead of `text`, or both.

`underline = int`:

The specified character (starting from 0) of the tab's text is underlined, negative means no underlining.

`sticky = string`:

The string consists of the usual combinations of 'n', 's', 'w', and 'e'. If the added widget is smaller than others, this is the side that it will be stuck to, and stretched if opposite sides are included, as usual.

`padding = dimension`:

Extra space added on all four sides outside the to-be-added widget.

`nb.insert(when, widget, option = value, option = value, ...)`

This is exactly the same as `add`, the same options and everything, but this lets you specify what position this tab should occupy. `when` can be an int, 0 for before the currently first tab, or an existing tab's widget object, in which case the insertion is just before that tab, or "last" to mean at the end.

`nb.hide(tab)`

`nb.forget(tab)`

These are the methods that are usually called `remove` and `forget`. `hide` makes the tab go away, but if it is added again without providing any options, it will be restored in its old position with all of its old settings. `forget` is mostly the same, but the tab's settings are not recorded, a later re-add would need to specify all of the options again. The removed tab can also be put back again by `insert`, but even if it was only hidden, all the options are required again. If you re-insert a hidden tab without any options, things go wrong: the system believes the tab has come back, but it does not become visible until you call `select` (below) to select it. At least, not on my Windows PC.

`nb.select()`

`nb.select(tab)`

The first form returns the almost useless name of the currently selected tab. The second form selects the given tab, as though the user had clicked on it.

`nb.enable_traversal()`

Special keyboard actions become possible. So long as the `Notebook` has focus, `control+tab` will select the tab immediately following the current one. `control+shift+tab` moves attention in the other direction, and if you used the `underline` option, `alt-q` will select the tab that has the letter `q` underlined.

```
nb.tab(tab)
nb.tab(tab, option = name)
nb.tab(tab, option = value, option = value, ...)
```

The first form returns a dictionary of all of the given tab's options as provided to add or insert and as modified by the third form of this method. The second form returns the value of the tab's option called `name`. Note that what appears to be an option is not. `option` is a keyword parameter. The third form changes the tab's options.

```
nb.index("end")
nb.index(tag)
```

The first form returns the number of tabs. The second form returns the position, starting from 0, of the given tag.

```
nb.tabs()
```

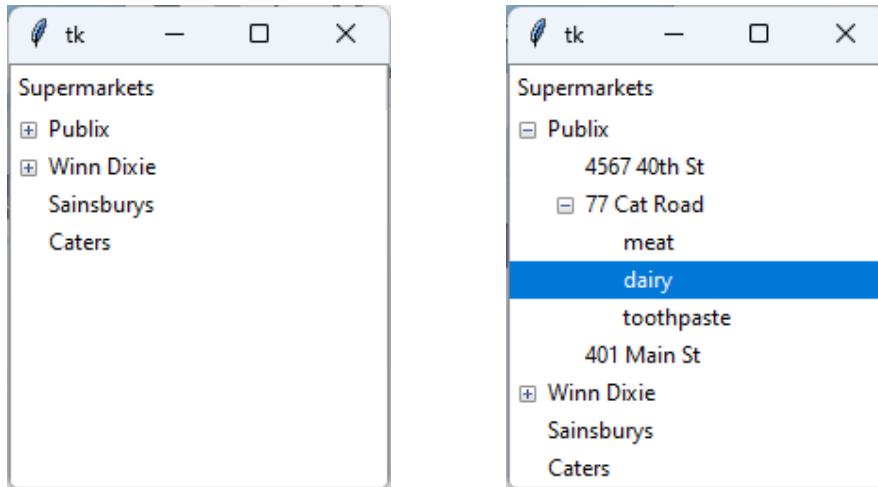
Returns a tuple containing the names of all of the tabs, in left-to-right order.

## 62. `ttk.Treeview`

A `Treeview` displays a hierarchical list of items. Generally, each item will have some text saying what it is, and a little plus sign in a box to its left. If the user clicks on the plus sign, the item opens up, displaying the sub-items that it contains, those sub-items can have their own sub-items and so on, going as deep as you want.

The example is for something that displays information about a number of supermarkets. At the top of the hierarchy are the names of the chain that the supermarket belongs to, below them are the individual shops, represented by their addresses, and below each of them are the various departments. Going any further would just make the example pointlessly big.





The left screenshot shows how things appear when the program starts. The right shows it after I clicked on the plus sign for Publix, then for its shop at 77 Cat Road, then on the dairy department. dairy has no little plus sign, I clicked on the text itself.

```

1 >>> tv = ttk.Treeview(win)
2 >>> tv.heading("#0", text = "Supermarkets", anchor = "w")
3
4 >>> pu = tv.insert("", "end", text = "Publix")
5 >>> wd = tv.insert("", "end", text = "Winn Dixie")
6 >>> sa = tv.insert("", "end", text = "Sainsburys")
7 >>> ca = tv.insert("", "end", text = "Caters")
8
9 >>> pu_1 = tv.insert(pu, "end", text = "4567 40th St")
10 >>> pu_2 = tv.insert(pu, "end", text = "77 Cat Road")
11 >>> pu_3 = tv.insert(pu, "end", text = "401 Main St")
12 >>> wd_1 = tv.insert(wd, "end", text = "88 Brook St")
13
14 >>> pu_2_m = tv.insert(pu_2, "end", text = "meat")
15 >>> pu_2_d = tv.insert(pu_2, "end", text = "dairy")
16 >>> pu_2_t = tv.insert(pu_2, "end", text = "toothpaste")
17
18 >>> tv.pack()

```

The call to `tv.heading` will be explained a little later. The constructor for a `Treeview` has the usual cursor and style options plus `height` which specifies the number of rows, not pixels. Vertical and horizontal `Scrollbar`s may be attached in the usual way. Other options are:

```

padding = all or
padding = (all, ) or
padding = (left_and_right, top_and_bottom) or
padding = (left, right, top_and_bottom) or
padding = (left, top, right, bottom)

```

Like the traditional `padx` and `pady`, but it allows the padding to be different on all four sides. I have given the padding values names that say which side they control. For example `left_and_right` gives the same padding at both the

left and right sides. Their values can be any kind of dimension, int for pixels, "0.25i", and so on.

`selectmode = "browse" or "extended" or "none"`

Controls how many rows the user may have selected at the same time. A row is selected by clicking on it, not its little plus sign, but the text. "browse" means that only one row may be selected. "extended", the default, means that any combination of rows may be selected at once. A normal click de-selects any currently selected rows and selects the one that was clicked instead. A control-click adds a row to the currently selected set. A shift-click selects every row from the first in the current selection up to the shift-clicked row. "none" means that selection by clicking is not possible, the "none" is there to display information, not to receive choices from the user. Clicking on the little plus signs still works of course.

These are most of the methods:

`tv.insert(parent, index, iid = None, option = value, option = value, ...)`

Creates a new row. parent indicates which existing row this one is nested under, it is the value that was returned when the parent was inserted. The new row becomes visible when its parent's little plus sign is clicked. The empty string means no parent, this row is at the top of the hierarchy.

index specifies the new row's position within its parent's underlings, 0 means it becomes the first, 2 means it is inserted before the second, and "end" means it goes at the end.

Every row has a special value that identifies it. If you want to choose what that value is, pass it in as the iid parameter. If you don't, insert will make up a unique value for it, and return it.

The options for insert are:

`text = string`

As in the example, this is the string that is displayed in the new row.

`image = PhotoImage`

Like with many widgets, you can have a graphical image instead of text. Note that there is no compound option, it is automatic. If you provide both text and image, the text will be to the right of the image, with no space between them.

When setting an image it is essential to remember, from the description of images in Canvasses, that your PhotoImage objects must all be stored securely in variables or lists or whatever for as long as the TreeView exists.

`open = True or False`

If set to `True`, this row will be created looking as though its little plus sign has already been clicked. Its underlings (if any) will also be visible. `False` is the default.

`tags = String or`

`tags = (String, String, ...)`

Tags are identifiers that many rows can share. There are methods that make the same thing happen to all rows that have a particular tag associated with them.

`tv.delete(iid, iid, iid, ...)`

`tv.detach(iid, iid, iid, ...)`

All of the rows identified by the `iids` (values returned by `insert`) are removed, along with all of their underlings. The `deleted` rows are destroyed, they can't be reinserted. `detach` is less destructive, allowing the rows to be put back later. This can only be done by using `move`, not `insert`.

`tv.move(iid, parent, index)`

The row identified by `iid` is detached from the view (unless it has already been detached), and re-inserted under the `parent` at the position given by `index`. Remember to use `""` for `parent` if it is to move to the top level and have no parent row.

`tv.see(iid)`

The row identified by `iid` is made visible by opening any unopen ancestors and scrolling if necessary.

`tv.get_children(iid)`

Returns a tuple of the names of all the direct children/underlings of the row identified by `iid` in the order that they are (or would be) displayed in. If you don't provide a parameter or it is the empty string, you will get all the top level rows.

`tv.set_children(newparent, iid, iid, iid, ...)`

All of the rows identified by the `iids` are removed from their current parent (if they have one), and become the children of row `newparent`. Any existing children of `newparent` are removed, but only by `detaching` them, so they can be put back by `move`.

`tv.item(iid)`

`tv.item(iid, string)`

`tv.item(iid, option = value, option = value, ...)`

The first form returns a dictionary of all the row's options (of the sort given to `insert`) and their values. The second form returns just the value of the one named option. The third form changes option values.

`tv.tag_configure(tagname)`

`tv.tag_configure(tagname, string)`

`tv.tag_configure(tagname, option = value, option = value, ...)`

Much like `item`. The first form returns a dictionary of all the settings that have been assigned to the tag. The second form returns the current value of the named option for that tag. The third form changes the options for all rows with a matching tag. The only options that can be set by this method are `foreground`, `background`, `font`, and `image`. Trying to set `text` fails silently.

`tv.tag_has(tagname)`

`tv.tag_has(tagname, iid)`

The first form returns a tuple of the iids for all rows that are controlled by the given tag. The second form returns `True` or `False` depending on whether the given row is controlled by the given tag.

`tv.tag_bind(tagname, event, callback)`

Every row controlled by the tag is set so that if the named event happens to it, or while it has focus, the callback function will be called with a descriptive event object saying what happened. The event can be just about any of the events listed in the subsection on binding to mouse and keyboard events.

`tv.selection_set(iid, iid, iid, ...)`

All currently selected rows are de-selected, and the specified rows are selected instead.

`tv.selection_add(iid, iid, iid, ...)`

The specified rows become selected, adding to any current selections.

`tv.selection_remove(iid, iid, iid, ...)`

Any of the specified rows that are currently selected are de-selected.

`tv.selection_toggle(iid, iid, iid, ...)`

If any specified row is selected, it becomes de-selected. If any is currently de-selected, it becomes selected.

`tv.focus()`

With no parameter, returns the iid for the item that “has focus”

`tv.focus(iid)`

With a parameter, moves the input focus to the given row, if this window had focus at all. That means that if any event has been `tag_bound` to that row, the callback function will be called if the event happens.

`tv.index(iid)`

Returns the given row’s position (0 for first child, etc) under its parent.

`tv.next(iid)`

Returns the iid for whatever row comes after the given iid under their parent, or the empty string if iid is its parent’s last child.

`tv.prev(iid)`

Returns the iid for whatever row comes immediately before the given iid under their parent, or the empty string if iid is its parent's first child.

`tv.parent(iid)`

Returns the iid for whatever row the given iid is a child/underling of, or the empty string if it is a top-level row.

`tv.exists(iid)`

True or False: is there a row with the given iid? detached rows count as existing, deleted ones don't.

`tv.identify_row(y)`

y is a pixel count, down from the top of the Treeview. Returns the iid for whatever row occupies that position, or the empty string if none do.

Each row in a Treeview isn't just limited to a little plus sign and some text. The entire Treeview can be given extra columns with their own headings, and the contents of any column for any row can be individually set to anything that can reasonably be printed. Unfortunately, only things that can be converted to text are allowed, no PhotoImages or widgets.

A Treeview may also display items of text data for each row, lined up in columns with headings. The only really essential thing for this is to use the TreeView's `set` method. If that is all you do, it will work, but it will not look very good.

`tv.set(iid, column, value)`

The data for the row and column given by iid and column is set to value. At the moment, column should be an int with zero representing the first data column. There is of course an extra column to the left of it, containing all the expandable labels, but that column is not included in the count. You can also give names to columns and use those instead of the int to make things more understandable. value is just about anything that can be rendered as text.

You give names to the columns by providing them in a list as the columns option for the Treeview's constructor, like this:

```
tv = ttk.Treeview(win, selectmode = "browse",
                  height = 8,
                  columns = ["pop", "are", "par"])
```

You *must* give names to all of the columns if you are going to have any at all.

By default, a row is set aside at the top of the Treeview to hold the column headings even if you don't have any headings. To get rid of it, you must add the option `show = "tree"`.

`tv.set(iid, column)`

Returns the data currently in the given cell.

`tv.set(iid)`

Returns a dictionary with the keys being the column names (not headings, but the names given to the `Treeview` constructor). The values are of course the corresponding data items.

`tv.column(column, option = value, option = value, ...)`

Configures a particular column, which may be an int or a column name. For this function the column name "#0" is used to refer to the leftmost part containing the expandable labels. The options are:

`width = pixels`

The default width is rather large, so you will probably want to set this.

`minwidth = pixels`

If you are not too picky you can just set a minimum acceptable width rather than demanding a set width.

`anchor = string`

The familiar "n", "nw", "w", "center", etc. to describe where data that is smaller than the cell size will be placed. The default is "w".

`stretch = 0`

Normally, when a `Treeview` is stretched horizontally, all of the columns grow wider. This says that this particular column will not join in, it will remain the same size.

`tv.column(column, option = which)`

Note that the word `option` is a keyword parameter, "which" should be the name of one of the options listed above. This form returns the current value of that option.

`tv.column(column)`

Returns a dictionary of all the option names and their values.

`tv.heading(column, option = value, option = value, ...)`

Gives the text and appearance of the column headings. Again, "#0" may be used for column. The options are:

`text = string`

The heading itself.

`image = PhotoImage`

A graphical image can be used as a column heading.

`anchor = string`

The familiar "n", "nw", "w", "center", etc. to describe where column heading is placed within the space available to it. The default is "w".

`command = string`

A callback function that will be called automatically if the user clicks on the column heading. No parameters are provided, so the only way to know which column heading was clicked is to use a different callback function for each of them, unless you remember about partial functions.

```
tv.heading(column, option = which)
```

Note that the word `option` is a keyword parameter, “which” should be the name of one of the options listed above. This form returns the current value of that option.

```
tv.heading(column)
```

Returns a dictionary of all the option names and their values.

```
tv.heading(column)
```

Returns a dictionary of all the option names and their values.

```
tv.identify_column(x)
```

`x` is a number of pixels from the left edge of the `Treeview`. Returns an identifier for which column is at that position. Strangely it does not return the names you gave with the constructor, the result is of the form “#0”, or “#1”, etc, where 1 refers to the first data column, or the empty string if the position is outside the `Treeview`.

```
tv.identify_region(x, y)
```

`x` and `y` are numbers of pixels from the top left corner of the `Treeview`. Returns a string saying what kind of thing is there:

“heading”: it is in amongst the headings,

“tree”: it is in one of the expandable labels,

“cell”: it is in one of the data cells,









“separator”: it is in the headings, but on top of one of the separators

“nothing”: it is outside the `Treeview`.

```
tv.bbox(iid, column = None)
```

If the row is visible this returns the tuple (`x`, `y`, `width`, `height`) giving the bounding box of the cell relative to the top left corner of the `Treeview`. If the row is not visible it returns the empty string. If the `column` parameter is left out you get the same but for the whole row. `column` may be an int data column number, a column name as given to the constructor, or “#0”, “#1”, etc.

Here is an example, and snippets of the code that produced it.

Compound Nations		Population	Area	Parts
	United Kingdom	68,138,484	93,628	4
	England	57,648,484	51.320	
	Wales	3,105,000	8.192	
	Scotland	5,480,000	30,977	
	Northern Ireland	1,905,000	5,530	
	United Arab Republic	132,480,251	461,000	2
	Egypt	109,546,720	390,121	
	Syria	22,933,531	71,500	

This got everything started:

```

1 >>> tv = ttk.Treeview(win, selectmode = "browse",
2 ...                       height = 8,
3 ...                       columns = ["pop", "are", "par"])
4 >>> tv.heading("#0", text = "Compound Nations", anchor = "w")
5 >>>
6 >>> tv.heading(0, text = "Population", anchor = "w", command = say)
7 >>> tv.column(0, width = 80)
8 >>> tv.heading(1, text = "Area", anchor = "w")
...

```

The expandable labels were produced like this:

```

21 >>> images = [ ]
22 ...
23 >>> def add(whr, txt, fnm):
24 ...     im = PhotoImage(file = "pictures\\" + fnm + ".png")
25 ...     images.append(im)
26 ...     return tv.insert(whr, "end", image = im, text = " " + txt)
27
28 >>> uk = add("", "United Kingdom", "UK")
29 >>> uar = add("", "United Arab Republic", "UAR")
30
31 >>> en = add(uk, "England", "E")
...

```

And the data was entered like this:

```

51 >>> def set(which, values):
52 ...     for index, value in enumerate(values):
53 ...         tv.set(which, index, value)
54
55 >>> set(uk, ["68,138,484", "93,628", 4])
56 >>> set(en, ["57,648,484", "51.320"])
57 >>> set(wa, ["3,105,000", "8.192"])
...

```

## 63. Menus

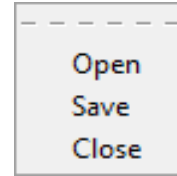


Menu bars may be added to the `Tk()` window and `toplevels` only. They appear as usual, at the top of the window, or on a Mac at the top of the screen. They are fairly easy to deal with. Tkinter calls normal menus “cascades” and also allows a few other things on a menu bar: a simple text string that can be clicked on, Checkbuttons, and Radiobuttons.

On a Mac, if you want the Menu bar to be at the top of the program’s window instead, these steps are supposed to work. First, pick any widget `wid` within the window, and say `win = wid.wininfo_toplevel()`. That retrieves the object that represents the top level window. Then create your Menu bar with `win` as its owner: `mb = Menu(win)`. That should be all it takes. Also on a Mac, only ordinary Menus may be added to the top Menu bar. Commands, Checkbuttons, and Radiobuttons may only be added to drop-down Menus.

To add menus to a window, first create a Menu object belonging to the window, then use the window’s `configure` method to tell it about it. Then create other Menu objects, this time belonging to the original Menu. Tkinter uses the Menu type for menu bars, actual menus, and even menus within menus. Finally give each of your Menus whatever options you want them to have through their `add_command` methods.

```
1 >>> from tkinter import *
2 >>> import functools as ft
3
4 >>> def action(which):
5 >>>     print(which, "was selected")
6
7 >>> win = Tk()
8
9 >>> can = Canvas(win, width = 300, height = 100, background = "white")
10 >>> can.pack()
11
12 >>> menbar = Menu(win)
13 >>> win.config(menu = menbar)
14
15 >>> filmen = Menu(menbar, tearoff = 0)
16 >>> filmen.add_command(label = "Open",
17 ...                   command = ft.partial(action, "Open"))
18 >>> filmen.add_command(label = "Save",
19 ...                   command = ft.partial(action, "Save"))
20 >>> filmen.add_command(label = "Close",
21 ...                   command = ft.partial(action, "Close"))
22 >>> menbar.add_cascade(label = "File", menu = filmen)
23
24 >>> edmen = Menu(menbar, tearoff = 0)
25 >>> edmen.add_command(label = "Copy",
26 ...                  command = ft.partial(action, "Copy"))
27 >>> edmen.add_command(label = "Paste",
28 ...                  command = ft.partial(action, "Paste"))
29 >>> menbar.add_cascade(label = "Edit", menu = edmen)
```



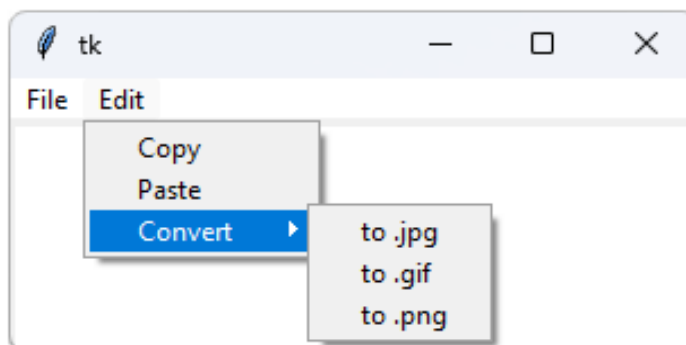
If I click on Save, “Save was selected” is printed. When creating one of the pull-down menus, failing to say `tearoff = 0` produces an ugly dotted line as you see in the picture to the right. I think this is supposed to allow the user to drag the menu away and put it somewhere else, but on my Windows PC it doesn't work. The `add_command` method needs a parameterless function as the callback, hence the use of `partial`.

If I change the end of the program to this, we'll see how easy it is to create a sub=menu.

```

29
30 >>> submen = Menu(edmen, tearoff = 0)
31 >>> submen.add_command(label = "to .jpg",
32 ...                     command = ft.partial(action, "jpg"))
33 >>> submen.add_command(label = "to .gif",
34 ...                     command = ft.partial(action, "gif"))
35 >>> submen.add_command(label = "to .png",
36 ...                     command = ft.partial(action, "png"))
37 >>> edmen.add_cascade(label = "Convert", menu = submen)
38 >>> menubar.add_cascade(label = "Edit", menu = edmen)

```



The `add_command` method can also be used on the main menu-bar `Menu` in exactly the same way as above. The result looks exactly the same as a cascade would, until you click on it. Nothing drops down, it just calls the callback function.

Some more `add_...` methods for `Menu` objects:

```
m.add_separator()
```

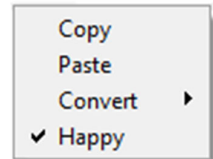


Produces a line at the next position, used to separate functionally different areas, or for whatever other reason you have.

```
1 >>> submen.add_separator()
```

`m.add_checkbutton(option = value, ...)`

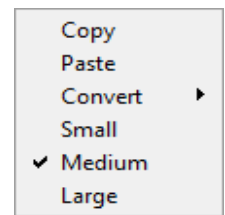
To add a Checkbutton to a Menu don't create a Checkbutton yourself, just use this method with a BooleanVar object as the variable option. It doesn't look like a normal Checkbutton, it is just another label, but every time you click on it the variable will flip its value, and when it is on a little tick appears to the left.



```
1 >>> b = BooleanVar()
2 >>> b.set(True)
3 >>> edmen.add_checkbutton(label = "Happy", variable = b)
```

`m.add_radiobutton(option = value, ...)`

Adding a set of Radiobuttons is almost the same, but in order to know which of the buttons is currently selected, you should create an IntVar or StringVar and provide it as the variable option, and decide which value it should be given for each button, providing that to the value option. It is still useful to set command to a callback function just so that you know when the selection has been changed. The initial value of that variable determines the initial selection, if any.



```
1 >>> i = IntVar()
2 >>> i.set(2)
3
4 >>> edmen.add_radiobutton(label = "Small", variable = i, value = 1)
5 >>> edmen.add_radiobutton(label = "Medium", variable = i, value = 2)
6 >>> edmen.add_radiobutton(label = "Large", variable = i, value = 3)
```

Checkbuttons and Radiobuttons may both be added to the main menu bar, but at least on my Windows PC, it doesn't work as we would expect. Clicking one of them still changes the variable (if there is one) and calls the callback function, but the appearance of the button does not change. No little tick or anything. So the user can not tell what is selected.

The Menu constructor has the normal options background or bg, foreground or fg, borderwidth or border or bd, disabledforeground, font, cursor, relief, activebackground, activeforeground, and activeborderwidth, but oddly no state, so disabledforeground is fairly pointless. These all apply to the dropped-down Menu itself, not to the title in the menu bar.

Other constructor options follow, but keep in mind that these are options for creating whole Menus, they are not the options for the `add_...` methods.

`postcommand = callback`

This parameterless callback function will be called whenever the Menu drops down, or expands, as the result of a mouse click on its title. On my Windows PC, I'm sure that the way this option behaves is not what was intended. Adding a `postcommand` for one single Menu results in that callback being called whenever any of the other Menus drop down too. If I provide one `postcommand` for one Menu, and another `postcommand` for another Menu, both get called whenever any Menu drops down as the result of a mouse click.

Also, `postcommands` happen only if a mouse click was involved. Menus tend to drop down whenever the mouse hovers above them and that has no effect.

This means that you can't tell whether a Menu has dropped down at all.

`selectcolor = colour`

This is the colour that the little tick for any Checkbuttons or Radiobuttons in this Menu will have when they are selected.

`tearoffcommand = callback`

`title = string`

These only apply if you have a `tearoff` enabled Menu. The callback function should be called when the Menu is torn off, and the torn off Menu's little window should have the given string as its title, but that just doesn't do anything on my Windows PC.

The `add_...` methods used to add choices to a Menu have their own set of options, as well as the usual `font`, `state`, and `underline`:

`background = colour`

`foreground = colour`

These over-ride the `background` and `foreground` settings given to the whole Menu for this item only. They may *not* be abbreviated in this use. These colours are used when the mouse is *not* over the item. When the mouse is over an item, the colours used are system dependent and can not be changed by any options.

`image = PhotoImage`

`compound = "left" or "right" or "top" or "bottom"`

These only work inside pull-down menus, not on the menu bar, at least on my Windows PC. These options are as usual if you want to have an image instead of or as well as a text label. There is one irregularity, at least on my Windows PC: if you set `compound` to `"left"` and it is an ordinary Menu item, not a Checkbutton or Radiobutton, the image (and text if any) will move further left than usual, and occupy the space used for Checkbuttons' and

Radiobuttons' little ticks if you have any. If you select "right", this doesn't happen.

`selectimage = PhotoImage`

This has no effect unless you also provide an image option. The `selectimage` is shown instead of the normal image when the item is selected, so it only applies to `Checkbuttons` and `Radiobuttons`.

`onvalue = value`

`offvalue = value`

Only for `Checkbuttonss`. Normally the controlling variable is set to 1 for selected and 0 for not selected. These options over-ride that.

`columnbreak = 0 or 1, the default is 0.`

Menu items are normally arranged entirely vertically. Setting `columnbreak` to 1 makes it start a new column of items before this one is added.

`accelerator = string`

Used to inform the user of "hot keys", keyboard shortcuts that have the same effect as selecting this item. The string you provide, something like "`^S`" as a keystroke equivalent for "Save", appears beside the label or image. This only works for items in drop-down `Menus`, not in the menu bar. It also does nothing to make the accelerator do its job. You would also have to make a keyboard event binding for the window.

`hidemargin = 0 or 1, the default is 0.`

This has absolutely no effect on my Windows PC, and causes an exception if I try to use it on a separator. Normally menu items have a small space between them where the background colour shows through. Setting `hidemargin` to 1 is supposed to result in that gap being left out. The idea is that you would use this for images that you want to run together.

Menu methods: These are for whole `Menus`, not `Menu` items.

`m.insert_... (position, ...)`

This is a set of five methods that have the same names (after the underline) as the `add_...` methods. They do exactly the same thing, except that instead of adding the new item to the end of a `Menu`, they insert them between or above existing `Menu` items. A position of 0 means that this becomes the new first item, 1 means the new second, and so on.

`m.delete (position)`

`m.delete (position1, position2)`

Positions are ints, 0 means the first `Menu` item, 1 the second, and so on. The first form removes one item from the `Menu`. The second form removes all items between the two positions, inclusive.

`m.type (position)`

Returns one of "cascade", "command", "checkboxbutton", "radiobutton", "tearoff", or "separator", depending on what kind of item is at that position in the Menu. If position is too big, it just goes for the last item, but if position is negative you get an exception.

`m.index("end")`

Returns the number of items in the Menu. Oddly, if the number is zero it returns None instead.

`m.entrycget(position, optionname)`

`m.entryconfigure(position1, option = value, option = value, ...)`

These let you see or change the options that were (or could have been) provided when the item was added or inserted.

`m.invoke(position)`

Simulate a mouse click on the indicated item, 0 for the first item in the Menu and so on. If it is a Radiobutton, it is set to selected. If it is a Checkbutton, its selectedness is reversed. All items have their callback function called.

`m.post(x, y)`

A detached but fully functional copy of the menu appears at the given position measure in pixels from the top left corner. It disappears again as soon as the user clicks the mouse within it. The position is supposed to be relative to the main window, but on my Windows PC it is relative to the top left corner of the primary screen.

`m.yposition(position)`

Is supposed to return the number of pixels between the top of the Menu and the top of the indicated item (or where it would be if it were open), so that you can use `post` to position a separated Menu immediately next to a particular Menu item. Unfortunately this really does measure from the top of the Menu, so on a Windows PC where `post` measures from the top of the screen, it is not very helpful.

## 64. Menubutton and Optionmenu

Menubuttons are closely related to Menus, they provide a different way to make the choices available. A Menubutton is an ordinary widget just like a Button that can be positioned anywhere you want. When clicked, it expands into a normal drop-down Menu.

Create a Menu exactly as in the previous subsection. Create a Menubutton and put it wherever you want. But the Menu needs be created with the Menubutton as its container, and the Menubutton needs to be created with the Menu as its menu option. The way around this is to create the Menubutton first, without giving it any menu option, then create the Menu, then use the Menubutton's `configure` method to set its menu option.

```

1 >>> win = Tk()
2
3 >>> can1 = Canvas(win, width = 150, height = 40, background = "white")
4 >>> can1.pack()
5 >>> mbt = Menubutton(win, text = "Colours", relief = "raised")
6 >>> mbt.pack()
7 >>> can2 = Canvas(win, width = 150, height = 40, background = "white")
8 >>> can2.pack()
9
10 >>> men = Menu(mbt, tearoff = 0)
11 >>> men.add_command(label = "Red", command = selred)
12 >>> men.add_command(label = "Green", command = selgreen)
13 >>> men.add_command(label = "Blue", command = selblue)
14 >>> men.add_command(label = "Indigo", command = selindigo)
15 >>> mbt.configure(menu = men)

```



The window as first created is on the left, its appearance when the button has been clicked is at the right.

The `Menubutton` constructor has all of the options of a normal `Button` except for `command`, together with `menu` as described above. It is also lacking the `flash` and `invoke` methods. It has one additional option:

`direction = "left" or "right" or "above" or "below"`. The default is "below".

Which side of the `Menubutton` the `Menu` will be stuck to when it is opened.

An `OptionMenu` is a very similar thing, but it is designed to be very easy to use. All you can do is provide a `StringVar` to control it, and a list of plain old strings to be the choices. The label displayed will initially be the value of the `StringVar`, even if it doesn't match any of the choices. Once a choice has been clicked on, that is displayed as the label.

`OptionMenus` also have all the options of a `Button` to control their appearance, plus an `indicatoron` option. If `indicatoron` is 1, the default, then a little rectangle appears beside the label. If it is 0, it doesn't.

It also supports the usual `command` option. Here, the callback function should have one parameter, it will be the string that was selected.

```

1 >>> def click(s):
2 ...     print("selected", s)

```

```

3
4 >>> sv = StringVar()
5 >>> om = OptionMenu(win, sv, "Tiny", "Small", "Medium", "Large",
6 ...     "Enormous", command = click)
7 >>> sv.set("Select Size")
8 >>> om.pack()

```



The image at the left is of the initial appearance. At the right it is after I first selected Small the clicked again to make another choice but haven't yet made that choice.

## 65. Dialogues

A dialogue is a small window that pops up temporarily to show some information or ask a question. Tkinter has six built-in kinds of dialogues, three for asking simple questions, one for yes/no questions or displaying a small amount of information, one for selecting files, and one for selecting a colour. For all of those, the appearance is totally system-dependent. Anything else, you can make for yourself in a `Toplevel` window.

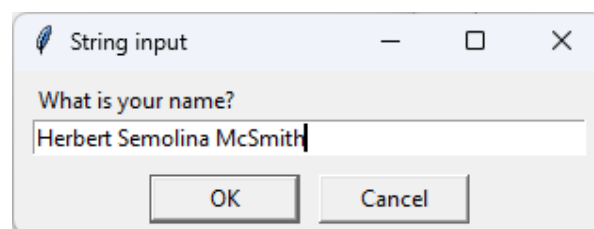
### i. Simpledialog

This class provides three class methods for very quickly getting an int, or a float, or a string from the user in response to a question.

```

1 >>> from tkinter import *
2 >>> from tkinter import simpledialog
3
4 >>> simpledialog.askstring("String input",
5 ...     "What is your name?\t\t\t\t")
6     'Herbert Semolina McSmith'

```





The tabs at the end of the question string are because the natural size of the dialogue window is too small for useful string input. A long question stretches the window, and the text input area grows with it. If the user clicks cancel or just closes the window, the return value is `None`.

There are four options:

```
initialvalue = string
```

A default value that will already be in the input area when the dialogue opens.

```
minvalue = string
```

```
maxvalue = string
```

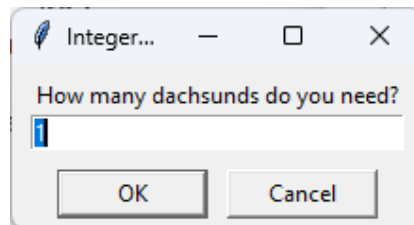
If the user enters a string below the minimum value or above the maximum value an error message is produced and the user gets another try. The comparisons are made the way strings are always compared, and are case sensitive. That makes these options almost useless, but they already were in the case of strings: when do you ever have a minimum dictionary ordering value for acceptable strings?

```
parent = window
```

The dialog will appear directly on top of the given window.

Two more class methods, `askinteger` and `askfloat`, work in exactly the same way. They only allow values of the right type, and `initialvalue`, `minvalue`, and `maxvalue` must also be of an appropriate type. Both return whatever value the user entered, or `None` if the user clicked cancel or closed the dialogue window.

```
1 >>> simpdialog.askinteger("Integer input",
2 ...     "How many dachunds do you need?",
3 ...     initialvalue = 1,
4 ...     maxvalue = 5)
5     1
```



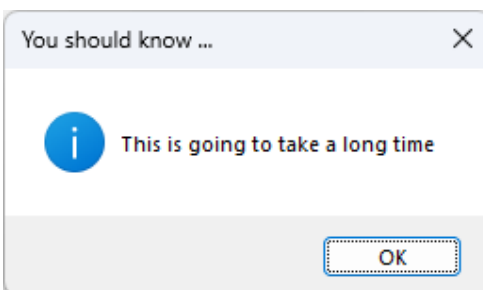
## ii. MessageBox

This kind of dialogue is for showing information or asking very basic questions. It has seven varieties, but you don't get many options. Each variety has a predefined appearance, number of buttons, and button labels. The documentation refers to these by the name `tkMessageBox`, but Python itself doesn't recognise that name. It only works if you use `messagebox` instead. All are created by using `messagebox` class methods, they all take a title to go in the pop-up window's title bar, and the

text to be displayed (both strings), and a few options. The text string may contain `\n` characters if you want it to span multiple lines.

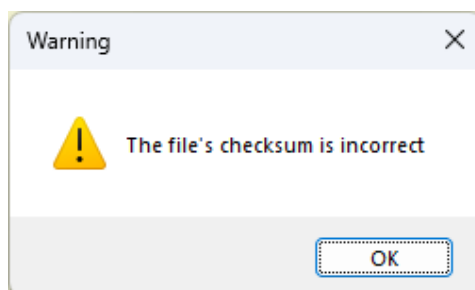
`showinfo`, `showwarning`, and `showerror` are all very similar:

```
1 >>> from tkinter import messagebox
2 >>> messagebox.showinfo("You should know ...",
3 ...                     "This is going to take a long time")
4     'ok'
```

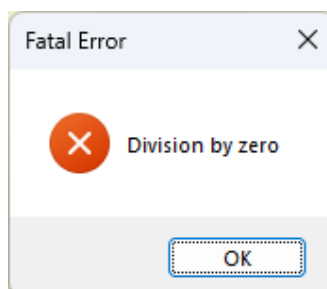


The function does not return until the user clicks the button, and it always returns the string `"ok"`. Even if the user closes the pop-up rather than clicking OK, it still returns `"ok"`.

```
1 >>> from tkinter import messagebox
2 >>> messagebox.showwarning("Warning",
3 ...                       "The file's checksum is incorrect")
4     'ok'
```



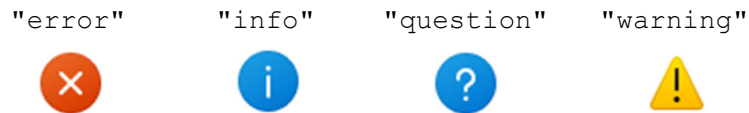
```
1 >>> from tkinter import messagebox
2 >>> messagebox.showerror("Fatal Error",
3 ...                     "Division by zero")
4     'ok'
```



All messagebox types have the following two options:

`icon = string`

Changes the icon that appears beside the text. Not as useful as one might imagine. You can't provide your own image, the value must be one of four strings, shown with their Windows PC results here:

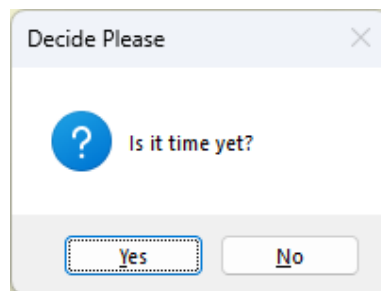


```
parent = window
```

Normally the pop-up appears on top of the main `Tk()` window. If you want it to appear over a `Toplevel` window, provide that object here.

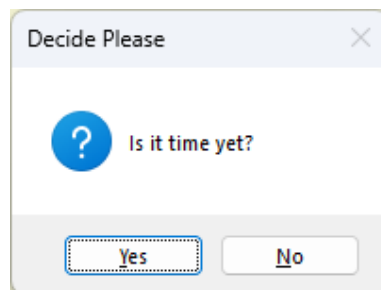
`askyesno`, `askquestion`, `askokcancel`, and `askretrycancel` provide a binary choice:

```
1 >>> from tkinter import messagebox
2 >>> messagebox.askyesno("Decide Please",
3 ...                     "Is it time yet?")
4     True
```



The return value is `True` if `Yes` is selected, `False` if `No`. The user does not get the option of closing the pop-up without answering.

```
1 >>> from tkinter import messagebox
2 >>> messagebox.askquestion("Decide Please",
3 ...                        "Is it time yet?")
4     'yes'
```



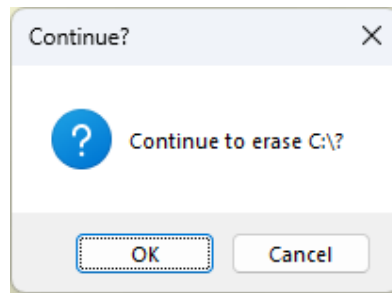
The appearance is identical to that of an `askquestion` on my Windows PC. The only difference is that the return values are `"yes"` and `"no"`.

```
1 >>> from tkinter import messagebox
```

```

2 >>> messagebox.askokcancel("Are you sure?",
3 ...     "Continue to erase C:\\?")
4     False

```



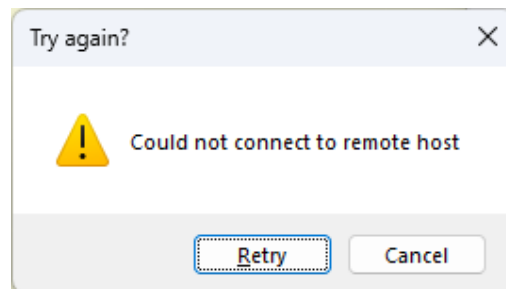
We are back to `True` and `False` as the return values. This one can be closed, in which case the result is `False`.

```

1 >>> from tkinter import messagebox
2 >>> messagebox.askretrycancel("Try again?",
3 ...     "Could not connect to remote host")
4     True

```

Again `True` and `False` are the return values. This one can also be closed.



The four binary choice dialogues accept a third option called `default`. Normally, the default button, the one that is pseudo-clicked if the user presses enter, is the left-most button. `default` changes this. There are five strings with the obvious meanings: "yes", "no", "ok", "retry", and "cancel". The documentation claims a sixth choice, "ignore", but that just causes an exception on my Windows PC.

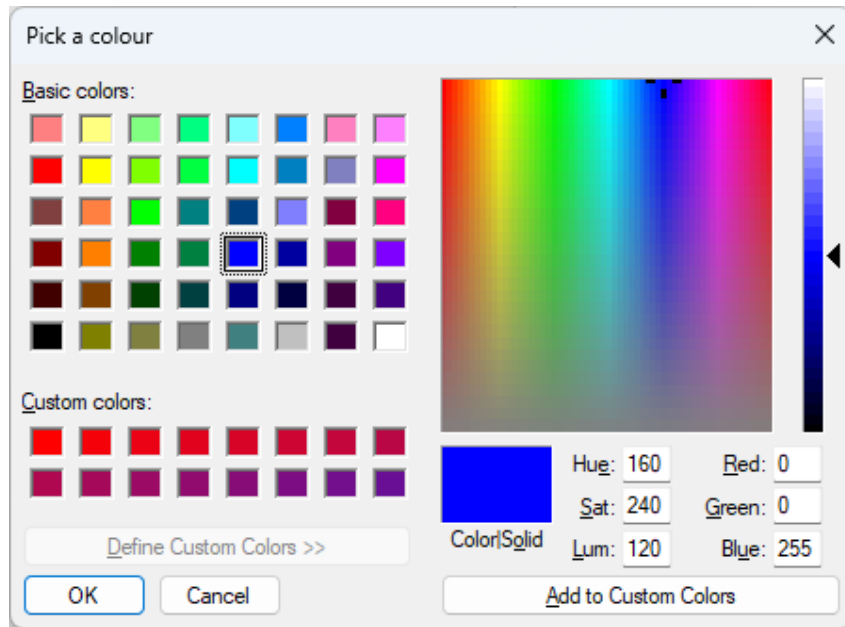
### iii. Colorchooser

As the name suggests, this dialogue lets the user select a colour for something. The first parameter is the colour that is initially chosen when the pop-up appears. The only other options are `title` and `parent`. The `title` of the pop-up may only be given as an option, and the `parent` option means the same as for a `messagebox`.

```

1 >>> from tkinter import colorchooser
2 >>> colorchooser.askcolor("white", title = "Pick a colour")
3     ((0, 0, 255), '#0000ff')

```



If the user does pick a colour and click OK, the result is a two-tuple. First a three-tuple of the 8-bit (red, green, blue) values, and second the hexadecimal version of that as a string. If the user clicks cancel or closes the dialogue, the result is `(None, None)`.

#### iv. Filedialog

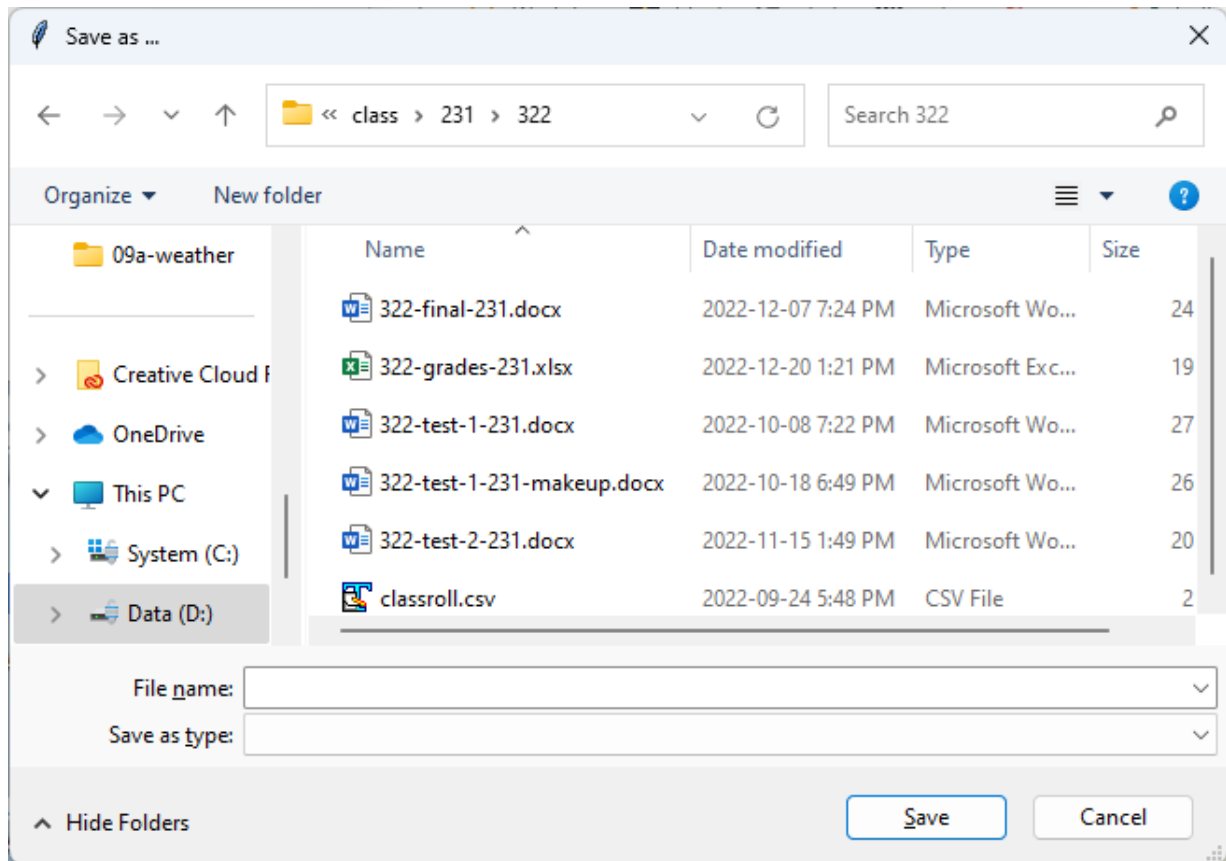
As expected, this dialogue allows the user to browse through discs and folders to select a file. It has two varieties, one for selecting a file to be read, one for selecting a file to be overwritten or created.

```

1 >>> from tkinter import filedialog
7 >>> filedialog.asksaveasfilename(title = "Save as ...",
9 ...                             defaultextension = ".txt",
10 ...                            initialdir = "D:\\Python")
11 >>> 'D:/class/231/322/classroll.csv'
```

The title option is, as usual, the title that appears at the top of the pop-up. `initialdir` overrides the default for which directory or folder should be shown at the beginning. If the user chooses a file either by double-clicking it, or selecting it and then clicking Open, the complete path for that file is returned as a string. If the user clicks Cancel or just closes the dialogue, the result is an empty string. The user may type a file name instead of browsing for one. If the user double-clicks on a file, or types in the name of a file that already exists, a warning and opportunity to change their mind appears.

If the user types in a filename that has no extension, the `defaultextension` string will be appended to it in the string that is returned, but it never appears in the dialogue.



Using `askopenfilename` instead of `asksaveasfilename` is almost the same, but the pop-up window does not have the "Save as type" input area, and of course the `defaultextension` option is ignored. This time, if the user selects a file that does not exist, it is rejected.

Other options:

`initialfile = string`

The string will appear as the initial contents of the "File name" input area.

`filetypes = list or tuple`

This gives the content of the "Save as type" input area with `...saveas...` and makes an extra input area appear with `...open...`, at least on my Windows PC.

The items in the list or tuple must themselves be two-tuples whose first is a descriptive string for a type of file, such as "python files", or "all files".

The second item in the tuples is either a single string or a tuple of strings giving the pattern or patterns for matching file names, such as `*.py`, `*.*`, or `(*py, *.py3)`.

Filling the gaps in the above example:

```
2 >>> types = (("python files", (*.py, *.py2, *.py3)),
```

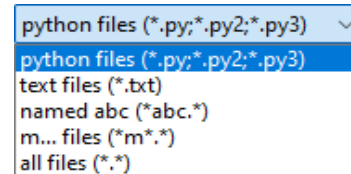
```

3 ...         ("text files", "*.txt"),
4 ...         ("named abc", "abc.*"),
5 ...         ("m... files", "m*.*"),
6 ...         ("all files", "*.*"))

8 ...         filetypes = types,

```

has this effect. Only the top line appears in the input area until I click on it, when this appears. Notice that two of my patterns, "abc.\*" and "m\*.\*", are not as I specified, an extra star has appeared at the beginning of both. What actually happens doesn't agree with either.



The idea is that only files whose names match the currently selected pattern, along with all folders, will be shown. The "m\*.\*" pattern as changed to "\*m\*.\*" does indeed show every file that has an "m" in its name before the dot. But the "abc.\*" pattern as changed to "\*abc.\*" completely misses two files called xxabcxx.txt and abc3.txt. What could be going on? Perhaps it is just another Windows PC peculiarity.

parent = window

The dialogue will appear directly on top of the given window.

typevariable

This option exists, something vaguely says that it has something to do with filters, but I can find no adequate description of it anywhere.

multiple = True or False

The documentation claims that this allows more than one file to be selected, but on my Windows PC using it just causes an exception.

Two more functions, `asksaveasfilenames` and `askopenfilenames` (both with an extra "s" at the end) are exactly the same, except that they allow multiple files to be selected. The return value is a tuple of the selected file names, but still just an empty *string* if cancel is clicked or the dialogue is closed.

If the `...name...` part of the four functions is left out (e.g. `askopenfile`), they open the selected file, in the appropriate read or write mode, and return those file objects in place of the names.

The `askdirectory` function only displays folders or directories, and naturally only allows folders to be selected. The `filetypes` option becomes meaningless, and using it causes an exception.

## 66. Pillow - better image processing

Pillow is an updated version of PIL, the Python Imaging Library. It supports many more graphics file formats than tkinter, and allows you to modify images in a way that would otherwise require a lot of programming.

Each version of Pillow can only run on the few most recent versions of Python, so you may need to update your Python software if it is a bit old.

Pillow is not part of the normal Python distribution, it needs to be downloaded and installed exactly as at the end of the section on Packages and Modules. Oddly, what you must install is `pillow`, but what you `import` is `PIL`. Or more likely, you will `import` something from `PIL`.

An introductory example will show how easy a lot of operations are. I'll open a jpeg file, display it, shrink it to half size, and save it as a png file.

```
1 >>> from PIL import Image
2
3 >>> full = Image.open("images\\10shilling.jpg")
4 >>> full.show()
5 >>> half = pic.resize((pic.width // 2, pic.height // 2))
6 >>> half.show()
7 >>> half.save("images\\half.png")
```



`open` only has one useful parameter, the image file. The other parameter, called `mode` has a default value of `"r"`, and giving it any other value causes an exception. The image file may be a normal file name in a string, or a `pathlib.Path` object, or an already open file object as returned by `open()`; it must be opened with mode `"rb"`.

`show` doesn't use any of Python's graphical abilities such as `Canvas` or `turtle`. It uses your computer's default viewer for the image type. `show`'s only option is `title`, giving the string that should appear in the window's title bar, but it usually has no effect.

Note that `resize` was just given a tuple as its single parameter. The result of the `resize` method is a new `Image` object. If you only want to resize a part of the original image, set the `box` parameter to a four-tuple `(x, y, w, h)` specifying the desired part as the pixel coordinates of its top left corner and its width and height.

If the size you give to `resize` does not have the same ratio of width to height as the original image, it will be distorted, so be careful. The `thumbnail` method (same parameters as `resize`) does not distort, it treats the width and height that you



supply not as required values, but as maxima. One will be the exact size, but the other will be changed to keep the same proportion. Unfortunately `thumbnail` is only willing to shrink images, it does not accept sizes larger than the existing ones.

When resizing, a rectangular neighbourhood of the original pixels may contribute to each new pixel with different weights. Two optional parameters give information about how this is to be handled. To understand these parameters requires more knowledge of graphics processing than is reasonable to expect in this context. Here I'll just give the names. `reducing_gap` should be a float  $> 1$ . `resample` may be any one of `Image.Resampling.BICUBIC`, `...BILINEAR`, `...BOX`, `...HAMMING`, `...LANCZOS`, or `...NEAREST`.

The `save` method may be given the file to save in any of the three ways that `open` accepts. If you provide an already open file (mode must be `"wb"`), `save` will not use the file's name to determine the format, you need to provide a second parameter called `format`. It should be one of the capital letter strings like `"PNG"` shown in the results in the next example, but using `SAVE` instead of `OPEN`. `save` automatically converts the image to the format indicated by the file name or the second parameter. If both are provided, the second parameter has precedence.

`save` accepts any keyword parameters you care to give it, but to have any effect they must be from the set that is supported for the format you want. There are far too many possibilities to cover.

The documentation says that you can see the list of file formats that PIL understands through `PIL.features.pilinfo()`, but my Windows PC denies the existence of `PIL.features`. To find the true list, use this:

```
1 >>> from PIL import Image
2
3 >>> def list_types(d):
4 ...     trans = dict({})
5 ...     exts = Image.registered_extensions()
6 ...     for k in exts:
7 ...         trans[exts[k]] = k
8 ...     bigs = sorted(set(d.keys()) & set(trans.keys()))
9 ...     return [ big + "(" + trans[big] + ")" for big in bigs ]
10
11 >>> list_types(Image.OPEN)
12 ['BLP(.blp)', 'BMP(.bmp)', 'BUFR(.bufr)', 'CUR(.cur)',
13  'DCX(.dcx)', 'DDS(.dds)', 'DIB(.dib)', 'EPS(.eps)',
14  'FITS(.fits)', 'FLI(.flc)', 'FTEX(.ftu)', 'GBR(.gbr)',
15  'GIF(.gif)', 'GRIB(.grib)', 'HDF5(.hdf)', 'ICNS(.icns)',
16  'ICO(.ico)', 'IM(.im)', 'IPTC(.iim)', 'JPEG(.jpeg)',
17  'JPEG2000(.j2c)', 'MPEG(.mpeg)', 'MSP(.msp)', 'PCD(.pcd)',
18  'PCX(.pcx)', 'PIXAR(.pxr)', 'PNG(.png)', 'PPM(.ppm)',
19  'PSD(.psd)', 'QOI(.qoi)', 'SGI(.sgi)', 'SUN(.ras)',
20  'TGA(.tga)', 'TIFF(.tiff)', 'WEBP(.webp)', 'WMF(.emf)',
21  'XBM(.xbm)', 'XPM(.xpm)']
```

That is the list of file types that it can currently open and therefore be able to process. To find the list of file types that it can create and therefore save, replace the `Image.OPEN` in the function call with `Image.SAVE`. That list is shorter.

To incorporate a PIL Image into a tkinter object only requires one extra step. To illustrate, I'll put an Image in a Canvas and a Label.

```
1 >>> from PIL import Image
2 >>> from PIL import ImageTk
3 >>> from tkinter import Tk, Label, Canvas
4
5 >>> win = Tk()
6 >>> pic = Image.open("images\\10shilling.jpg")
7 >>> img = ImageTk.PhotoImage(pic)
8 >>> lab = Label(win, image = img)
9 >>> lab.pack()
10 >>> can = Canvas(win, width = 400, height = 400)
11 >>> can.pack()
12 >>> can.create_image(5, 5, anchor = "nw", image = img)
```

The example showed two of Image's attributes, `width` and `height`, and the use of `//` instead of `/` to divide by two illustrates the fact that Pillow only accepts exact ints as coordinates. There are some others:

```
img.size = (w, h)
```

The image size as a width, height two-tuple.

```
img.format = string or None
```

A string, like "PNG" or "JPEG". This is not what you might expect. File formats like png and jpeg are responsible for how colours are represented, how the image is compressed, and so on. Once an Image has been successfully opened or if it was created by manipulating another image, these details are meaningless, it is all just an uncompressed rectangle of pixel values. So most of the time, `format` will be `None`.

```
img.filename = string
```

The name of the file that open read to create the Image. If it didn't come from a file, the empty string is used.

```
img.mode = string
```

How colours are represented. Two possible values, "RGB" and "CMYK", are well known. The others require a lot of knowledge of image processing to understand.

```
img.palette = ImagePalette object
```

Some formats, such as gif, are limited to a small-ish number of different colours. The definitions of the chosen colours, possibly as RGB triplets, are stored in a list somewhere in the file. The image data refers to colours by their position in this list. The list is called a palette. This attribute is designed to give access to the palette, but at the time of writing it is still very experimental and lacking any details.

```
img.is_animated = True or False
```

```
img.n_frames = int
```

Some formats, such as gif, can be animated. Typically when displayed they will cycle through a loop of individual ordinary images. These attributes tell you a little bit about that.

Other useful methods that return new Images rather than altering existing ones:

```
img.crop((left_x, top_y, right_x, bottom_y))
```

The image is reduced in size by discarding everything outside of a given rectangle. The one parameter is a four-tuple with everything measured in pixels from the top left corner.

```
img.rotate(angle, expand = True or False)
```

The angle is degrees anticlockwise. If the image is square, and angle that is not a multiple of 90 will result in part of the image being lost. For oblong images, anything other than 180 degrees will cause image loss. The expand parameter, which is normally False can be used to change this. if expand is True the height and width will expand enough for the entire rotated image to fit.

```
img.rotate(kind)
```

This does operations that could be thought of as in some way flipping the image. kind is actually an int, but there are seven names for them. The effects of the first five are obvious from their names: Image.ROTATE\_90, Image.ROTATE\_180, Image.ROTATE\_270 (rotations are anticlockwise), Image.FLIP\_LEFT\_RIGHT, Image.FLIP\_TOP\_BOTTOM (mirror images of a kind). Image.TRANSPOSE is the same operation as transposing a matrix, y coordinates become x coordinates and x coordinates become y coordinates. It is equivalent to a FLIP\_LEFT\_RIGHT operation followed by a ROTATE\_90. Image.TRANSVERSE is very similar, being equivalent to a FLIP\_LEFT\_RIGHT operation followed by a ROTATE\_270.