Python:
Introduction for Programmers

Bruce Beckles

Bob Dowling

University Computing Service

Scientific Computing Support e-mail address:
escience-support@ucs.cam.ac.uk

1

This is the UCS one day course on Python for people who have some experience of programming in a programming language other than Python. The course assumes that you know *no* Python whatsoever – we warn all those people who already have some Python experience that they are likely to be bored on this course. Also, those people who already have considerable programming experience in several different programming languages are also likely to be bored, and would be better off just working through the notes in their own time, or looking at one of the many on-line Python tutorials.

Note that this course covers Python 2.2 to 2.6, which are the most common versions currently in use – it does **NOT** cover the recently released Python 3.0 (or 3.1) since those versions of Python are so new. Python 3.0 is significantly different to previous versions of Python, and this course will be updated to cover it as it becomes more widely used.

Also, people who do **not** already know how to program in another language – or who have minimal programming experience – and want to learn Python will probably find this course too fast/difficult for them. Such people are better off attending the UCS "Python: Introduction for Absolute Beginners" three afternoon course. For details of this course, see:
        http://training.csx.cam.ac.uk/course/python

The official UCS e-mail address for all scientific computing support queries, including any questions about this course, is:
        escience-support@ucs.cam.ac.uk
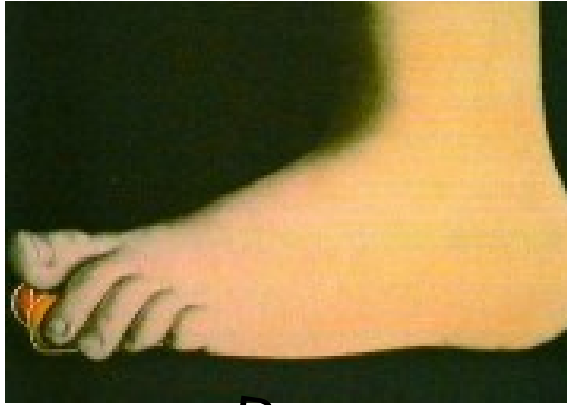
Python: Introduction for Programmers

Bruce Beckles Bob Dowling
University Computing Service
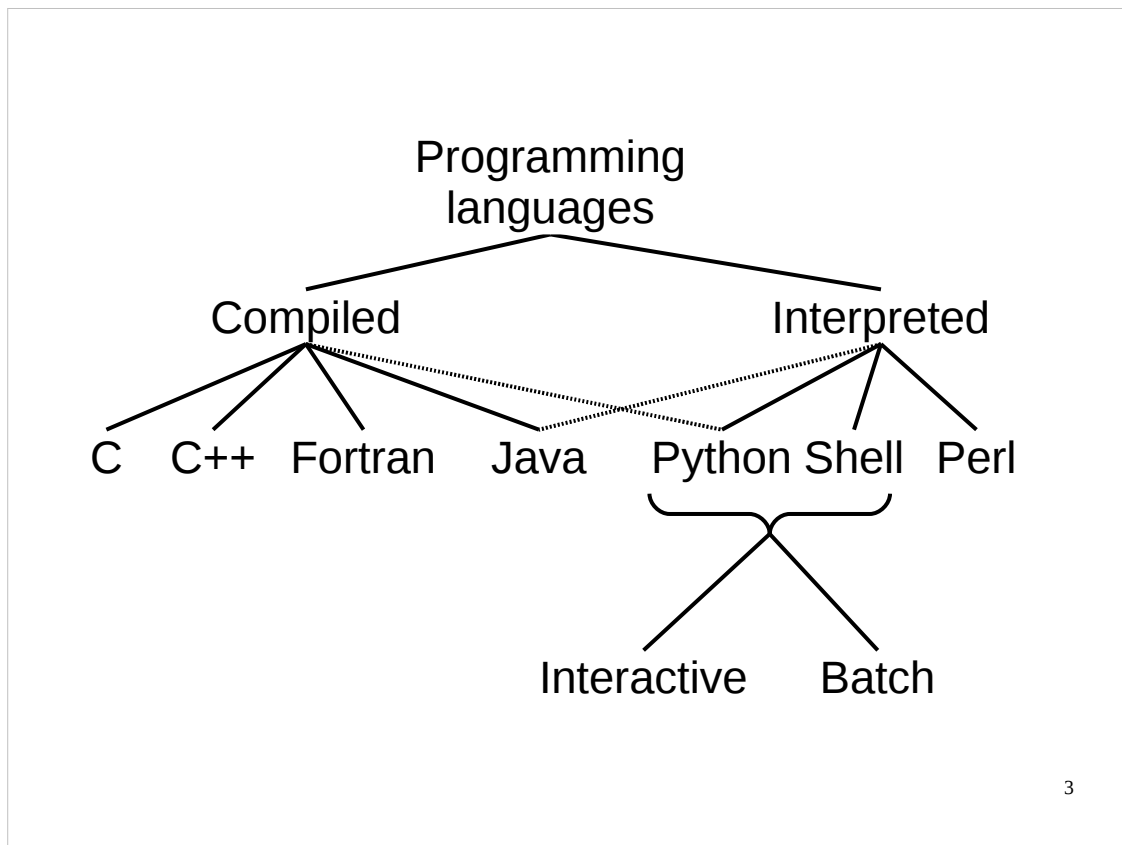Scientific Computing Support e-mail address:
escience-support@ucs.cam.ac.uk

Python is named after Monty Python's Flying Circus, not the constricting snake.

There are various versions of Python in use, the most common of which are releases of Python 2.2, 2.3, 2.4, 2.5 and 2.6. (The material in this course is applicable to versions of Python in the 2.2 to 2.6 releases.)

On December 3rd, 2008, Python 3.0 was released. Python 3.0 is significantly different to previous versions of Python, is **not** covered by this course, and *breaks* backward compatibility with previous Python versions in a number of ways. As Python 3.0 and 3.1 become more widely used, this course will be updated to cover them.

Programming
languages

Compiled   Interpreted

C   C++   Fortran   Java   Python Shell Perl

Interactive   Batch

3

As you probably know, programming languages split into two broad camps according to how they are used.

Compiled languages go through a "compilation" stage where the text written by the programmer is converted into machine code. This machine code is then processed directly by the CPU at a later stage when the user wants to run the program. This is called, unsurprisingly, "run time".

Interpreted languages are stored as the text written by the programmer and this is converted into machine instructions all in one go at run time.

There are some languages which occupy the middle ground. Java, for example, is converted into a pseudo-machine-code for a CPU that doesn't actually exist. At run time the Java environment emulates this CPU in a program which interprets the supposed machine code in the same way that a standard interpreter interprets the plain text of its program. In the way Java is treated it is closer to a compiled language than a classic interpreted language so it is treated as a compiled language in this course.

Python can create some intermediate files to make subsequent interpretation simpler. However, there is no formal "compilation" phase the user goes through to create these files and they get automatically handled by the Python system. So in terms of how we use it, Python is a classic interpreted language. Any clever tricks it pulls behind the curtains will be ignored for the purposes of this course.

So, if an interpreted language takes text programs and runs them directly, where does it get its text from? Interpreted languages typically support getting their text either directly from the user typing at the keyboard or from a text file of commands.

If the interpreter (Python in our case) gets it input from the user then we say it is running "interactively". If it gets its input from a file we say it is running in "batch mode". We tend to use interactive mode for simple use and a text file for anything complex.

Interactive use

$ **python**
Python 2.6 (r26:66714, Feb 3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on …
Type "help", "copyright", "credits" or "license" …

>>> **print 'Hello, world!'**
Hello, world!

>>> **3**
3

4

Now that we have a Unix command line interpreter we issue the command to launch the Python interpreter. That command is the single word, "python".

In these notes we show the Unix prompt, the hint from the Unix system that it is ready to receive commands, as a single dollar sign character ($). On PWF Linux the prompt is actually that character preceded by some other information.

Another convention in these notes is to indicate with the use of **bold face** the text that you have to type while regular type face is used for the computer's output.

The interactive Python interpreter starts by printing three lines of introductory blurb which will not be of interest to us.

After this preamble though, it prints a Python prompt. This consists of three "greater than" characters (>>>) and is the indication that the Python interpreter is ready for you to type some Python commands. You cannot type Unix commands at the prompt. (Well, you can type them but the interpreter won't understand them.)

So let's issue our first Python command. There's a tradition in computing that the first program developed in any language should output the phrase "Hello, world!" and we see no reason to deviate from the norm here.

The Python command to output some text is "print". This command needs to be followed by the text to be output. The text, "Hello, world!" is surrounded by single quotes (') to indicate that it should be considered as text by Python and not some other commands. The item (or items) that a command (such as print) needs to know what to do are called its "arguments", so here we would say that 'Hello, world!' is the print command's argument.

The command is executed and the text "Hello, world!" is produced. The print command always starts a new line after outputting its text.

You will probably not be surprised to learn that everything in Python is *case-sensitive*: you have to give the print command all in lower-case, "PRINT", "pRiNt", etc. won't work.

Note that what the Python interpreter does is evaluate whatever it has been given and outputs the result of that evaluation, so if we just give it a bare number, e.g. 3, then it evaluates that number and displays the result:
3

4

```
$ python
Python 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on …
Type "help", "copyright", "credits" or "license" …

>>> print 'Hello, world!'
Hello, world!

>>> 3
3

>>>
$
```

To quit the Python interpreter:
Press *control+d*

Unix prompt

Now that we are in the Python interpreter it would be useful if we knew how to get out of it again. In common with many Unix commands that read input from the keyboard, the interpreter can be quit by indicating "end of input". This is done with a "control+d". To get this hold down the control key (typically marked "Ctrl") and tap the "D" key once. Then release the control key.

Be careful to only press the "D" key once. The "control+d" key combination, meaning end of input, also means this to the underlying Unix command interpreter. If you press "control+d" twice, the first kills off the Python interpreter returning control to the Unix command line, and the second kills off the Unix command interpreter. If the entire terminal window disappears then this is what you have done wrong. Start up another window, restart Python and try again.

If you are running Python interactively on a non-Unix platform you may need a different key combination. If you type "exit" at the Python prompt it will tell you what you need to do on the current platform. On PWF Linux you get this:

```
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>>
```
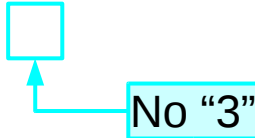
It would also be useful if we knew how to use Python's help system. We'll look at how we access Python's help in a few slides' time.

Batch use

```
#!/usr/bin/python

print 'Hello, world!'

3
```
                                                hello.py

$ **python hello.py**

Hello, world!

No "3"

Now let us look at the file `hello.py` in the home directory of the course accounts you are using. We see the same two lines:

```
print 'Hello, world!'
3
```

(Ignore the line starting #. Such lines are comments and have no effect on a Python program. We will return to them later.)

The suffix ".py" on the file name is not required by Python but it is conventional and some editors will put you into a "Python editing mode" automatically if the file's name ends that way. Also, on some non-Unix platforms (such as Microsoft Windows) the ".py" suffix will indicate to the operating system that the file is a Python script. (Python programs are conventionally referred to as "Python scripts".)

We can run this script in batch mode by giving the name of the file to the Python interpreter at the Unix prompt:

```
$ python hello.py
Hello, world!
```

This time we see different output. The `print` command seems to execute, but there is no sign of the 3.

We can now see the differences between interactive mode and batch mode:

- Interactive Python evaluates every line given to it and outputs the evaluation. The `print` command doesn't evaluate to anything, but prints its argument at the same time. The integer 3 outputs nothing (it isn't a command!) but evaluates to 3 so that gets output.
- Batch mode is more terse. Evaluation is not output, but done quietly. Only the commands that explicitly generate output produce text on the screen.
  Batch mode is similarly more terse in not printing the introductory blurb.

```
$ python
Python 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on …
Type "help", "copyright", "credits" or "license" …

>>> help
Type help() for interactive help, or help(object) for
help about object.
>>> help()

Welcome to Python 2.6!  This is the online help utility.

If this is your first time using Python, …

help>          ←          help utility prompt
```

Launch the Python interpreter again as we will be using it interactively for a while.

The first thing we will do is look at Python's interactive help (which Python refers to as its "online help utility").

You may have noticed that the introductory blurb we get when we start Python suggests a number of words we might like to type to get "more information". One of those words is "help". Let's see what happens if we type "help" at the Python prompt:

```
>>> help
Type help() for interactive help, or help(object) for help about object.
```

Python tells us there are two ways we can get help from within the Python interpreter. We can either get interactive help via its online help utility by typing "help()", which we'll do in a moment, or we can directly ask Python for help about some particular thing (which we'll do a bit later by typing "help('thing')" where "thing" is the Python command we want to know about).

So let's try out Python's interactive help, by typing "help()" at the Python prompt.

This will start Python's online help utility as shown above. Note that we get an introductory blurb telling us how the utility works (as well as how to quit it (type "quit")), and the prompt changes from Python's prompt ("">>>"") to:

```
help>
```

Note that the online help utility will only provide useful information if the Python documentation is installed on the system. If the documentation hasn't been installed then you won't be able to get help this way. Fortunately, the complete Python documentation (exactly as given by the online help utility) is available on the web at:

```
http://docs.python.org/
```

…in a variety of formats. It also includes a tutorial on Python.

Using Python's online help utility interactively is really straightforward: you just type the name of the Python command, keyword or topic you want to learn more about and press return. Let's see what happens if we ask it about the `print` command:

```
help> print
```

On PWF Linux the screen clears and we get a new screen of text that looks something like this:

```
    ----------------------------------------------------------------------


    6.6 The print statement

         print_stmt      ::=     "print" ([expression[1] ("," expression[2])* [","]
                         | ">>" expression[3] [("," expression[4])+ [","])


    Download entire grammar as text.[5]


    print evaluates each expression in turn and writes the resulting object
    lines 1-10
```

(*For space reasons only the first 10 lines of the help text are shown (in very small type – don't try and read this text in these notes here but rather try this out yourself on the computer in front of you*).) You can get another page of text by pressing the **space bar**, and move back a page by typing "**b**" (for "back"), and you can quit from this screen by typing "**q**" (for "quit"). (If you want help on on what you can do in this screen, type "**h**" (for "help").) The program that is displaying this text is not Python but an external program known as a *pager* (because it displays text a page at a time) – this means exactly which pager is used and how it behaves depends on the underlying operating system and how it is configured. Note that on PWF Linux when you finish reading this help text and type "q", the help text disappears and you get back the screen you had before, complete with the "`help>`" prompt.

When you've finished trying this out, quit the help utility by typing "`quit`" at the "`help>`" prompt and pressing return. Finally, we can also get help on a Python command, keyword or help topic directly, without using the online help utility interactively. To do this, type "`help('`*thing*`')`" at the Python prompt, where "*thing*" is the Python command, etc. on which you want help (note the quotes (') around "`'`*thing*`'`").

```
$ python
Python 2.6 (r26:66714, Feb  3 2009, 20:52:03)
[GCC 4.3.2 [gcc-4_3-branch revision 141291]] on …
Type "help", "copyright", "credits" or "license" …

>>> print 3                    Pair of arguments
3                              separated by a comma

>>>  print 3 , 5

3   5
                               Pair of outputs
                               no comma
```

9

Before we go any further, we'll demonstrate a property of the `print` command.

We have already seen `print` used with a single argument.

But we can also use it with two arguments, separated by a comma.

Note that `print` outputs the two arguments (separated by a single space), and not the comma. The comma separates the arguments; it is not an argument in its own right.

This is a general feature of Python commands: when the command has multiple arguments, commas are used to separate the arguments.

Using Python for science → *Science* → Quantitative → Numbers → Using numbers in Python
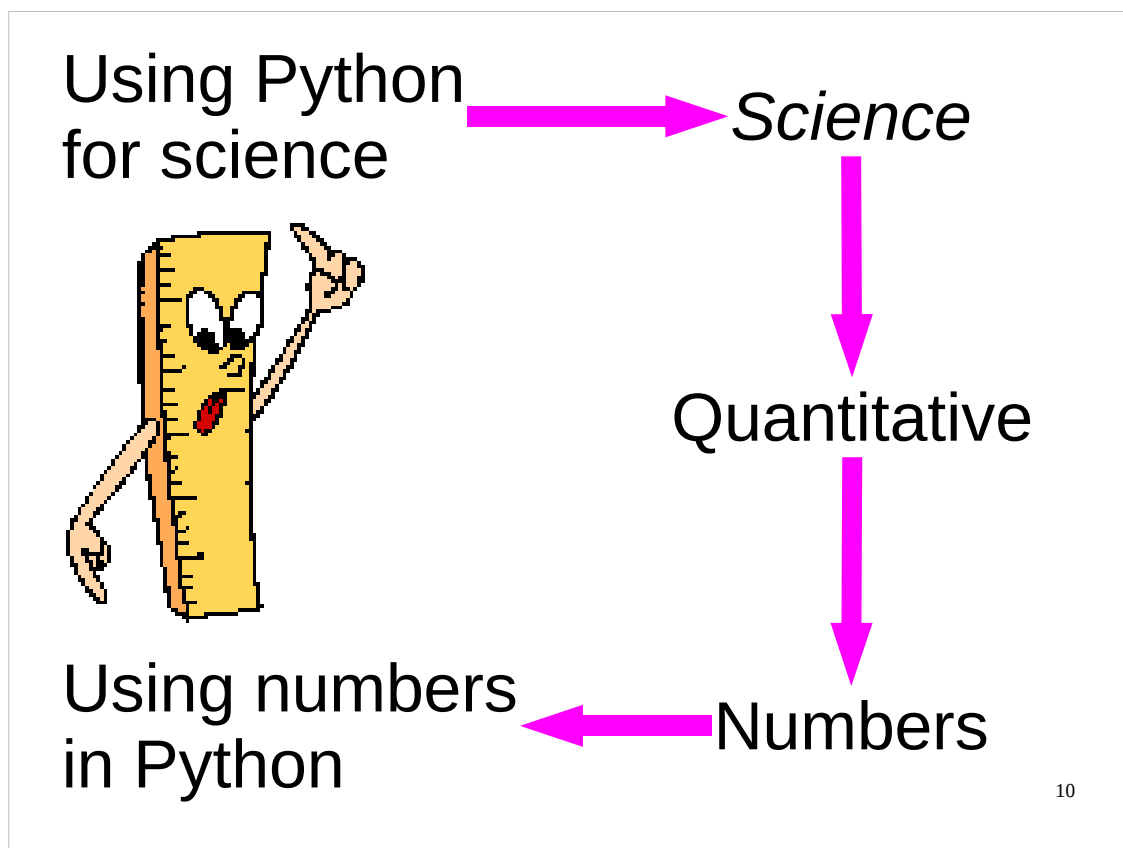
We are going to spend a little while talking about numbers and how computers (and in particular Python) handle them.

Why?

Well, this is a course in the "Scientific Computing" series and science is about quantifying things. That requires numbers to represent those quantities and so we need to understand how Python handles numbers.

Those of you who are already familiar with how computers handle numbers, please bear with us for a little while – despite what you may be tempted to think, this is not common knowledge to *all* programmers, especially those who have learnt to program in languages which tend to hide the subtleties of computer arithmetic from the programmer (e.g. Perl, Visual Basic).

$\mathbb{Z}$ Integers

$$\{ \ldots -2, -1, 0, 1, 2, 3, \ldots \}$$

We will start with the integers. (In case you have not met it before, $\mathbb{Z}$ is the mathematical symbol for the integers.)

```
>>> 7+3              >>> 7-3
10                   4

>>> 7*3              >>> 7**3        7³: use "**" for
21                   343            exponentiation

>>> 7/3              >>> -7/3        integer division
2                    -3             rounds down

>>> 7%3              >>> -7%3        remainder (mod)
1                    2              returns 0 or positive
                                    integer
```

12

On some systems (including PWF Linux) the Python interpreter has built-in command line history. If you press the up and down arrows you can navigate backwards and forwards through your recent Python commands. You can also move left and right through the line (using the left and right arrow keys) to edit it.

Most of you will be familiar with the basic arithmetic operations. (For those who have not met these conventions before, we use the asterisk, "*" for multiplication rather than the times sign, "×", that you may be used to from school. Similarly we use the forward slash character, "/" for division rather than the division symbol, "÷".)

The first thing to note, if you have not already come across it, is that division involving two integers ("integer division") always returns an integer. Integer division rounds *strictly* downwards, so the expression "7/3" gives "2" rather than "2 1/3" and "-7/3" gives "-3" as this is the integer below "-2 1/3". So (-7)/3 does not evaluate to the same thing as -(7/3). (Integer division is also called "floor division".)

There are also two slightly less familiar arithmetic operations worth mentioning:

* Exponentiation (raising a number to a power): the classical notation for this uses superscripts, so "7 raised to the power 3" is written as "7^3". Python uses double asterisks, "**", so we write "7^3" as "7**3". You are permitted spaces around the "**" but not inside it, i.e. you cannot separate the two asterisks with spaces. Some programming languages use "^" for exponentiation, but Python doesn't – it uses "^" for a completely different operation (bitwise exclusive or (*xor*), which we don't cover in this course). Python also has a function, pow(), which can be used for exponentiation, so pow(x, y) = x^y. E.g.
  ```
  >>> pow(7, 3)
  343
  ```
* Remainder (also called "mod" or "modulo"): this operation returns the remainder when the first number is divided by the second, and Python uses the percent character, "%" for this. "7%3" gives the answer "1" because 7 leaves a remainder of 1 when divided by 3. The remainder is always zero or positive, even when the number in front of the percent character is negative, e.g. "-7%3" gives the answer "2" because $(3 \times -3) + 2 = -7$.

Another function worth mentioning at this point is the abs() function, which takes a number and returns its *absolute value*. So abs(a) = |a|. (Note that pow() and abs(), in common with most functions, require parentheses (round brackets) around their arguments – the print function is a special case that doesn't.)

```
>>> 2*2
4

>>> 4*4
16

>>> 16*16
256

>>> 256*256
65536

>>> 65536*65536
4294967296L  ←————————  "large" integer
```

Python's integer arithmetic is very powerful and there is no limit (except the system's memory capacity) for the size of integer that can be handled. We can see this if we start with 2, square it, get and answer and square that, and so on. Everything seems normal up to 65,536.

If we square 65,536 Python gives us an answer, but the number is followed by the letter "L". This indicates that Python has moved from standard integers to "long" integers which have to be processed differently behind the scenes but which are just standard integers for our purposes. Just don't be startled by the appearance of the trailing "L".

Note: If you are using a system with a 64-bit CPU and operating system then the 4,294,967,296 also comes without an "L" and the "long" integers only kick in one squaring later.

```
>>> 4294967296*4294967296
18446744073709551616L

>>> 18446744073709551616 *
18446744073709551616
340282366920938463463374607431768211456L

>>> 2**521 - 1
686479766013060971498190079908139321726
943530014330540939446345918554318339765605
212255964066145455497729631139148085803
7121987999716643812574028291115057151L
```

No inherent limit to Python's integer arithmetic: can keep going until we run out of memory

14

We can keep squaring, limited only by the base operating system's memory. Python itself has no limit to the size of integer it can handle.
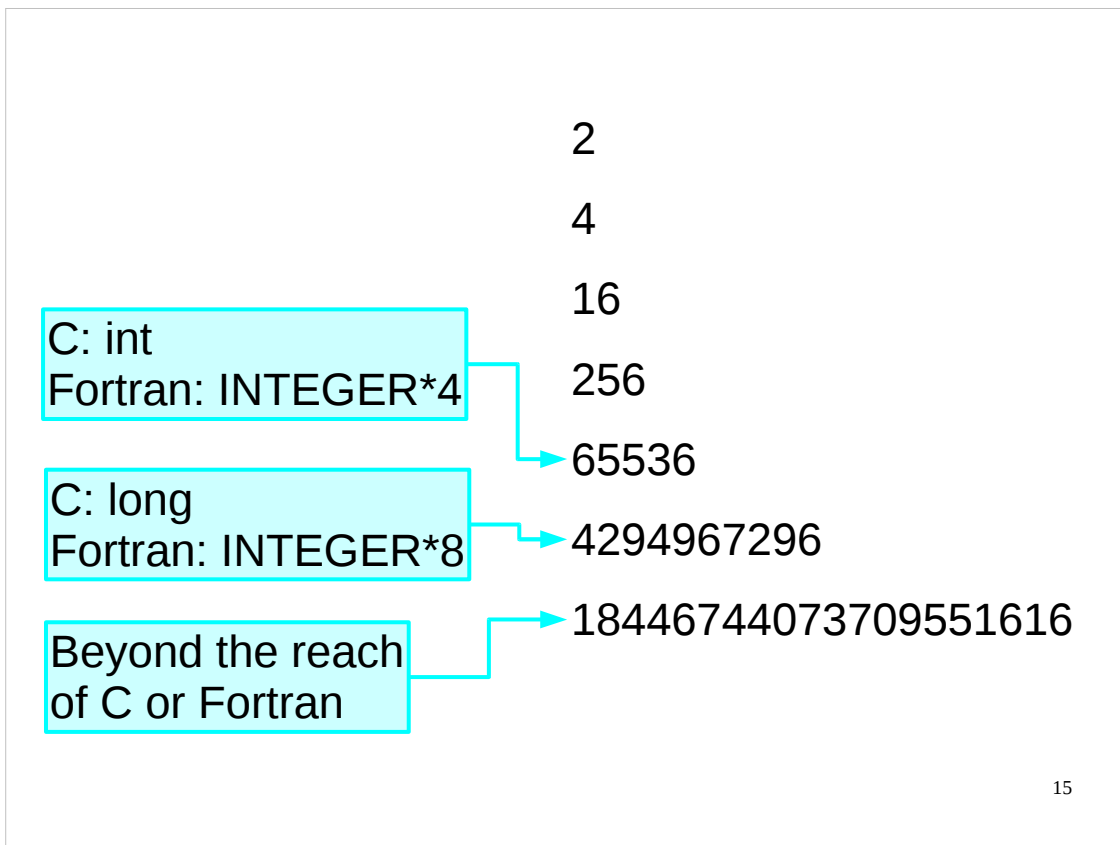
Indeed, we can try calculating even larger integers using exponentiation, e.g. "2**521 – 1" (2^{521}-1).

(In case you are wondering, "2^{521}-1" is the 13th Mersenne prime, which was discovered in 1952 by Professor R. M. Robinson. It has 157 digits. If you are curious about Mersenne primes, the following URLs point to documents that provide good overviews of these numbers:

http://primes.utm.edu/mersenne/

http://en.wikipedia.org/wiki/Mersenne_prime

)

2

4

16

256

C: int
Fortran: INTEGER*4

65536

C: long
Fortran: INTEGER*8

4294967296

Beyond the reach
of C or Fortran

18446744073709551616

15

As you've probably realised, Python is quite exceptional in this regard. C and Fortran have strict limits on the size of integer they will handle. C++ and Java have the same limits as C but do also have the equivalent of Python's "long integers" as well (although they call them "big integers"; note that for C++ you need an additional library). However, in C++ and Java you must take explicit action to invoke so-called "big integers"; they are not engaged automatically or transparently as they are in Python.

(Note that more recent versions of C have a "long long" integer type which you can use to get values as large as 18,446,744,073,709,551,615.)

Floating point numbers

And that wraps it up for integers.

Next we would like to move on to real numbers but here we encounter the reality of computing not reflecting reality. Subject to the size limits of the system memory we could say that the mathematical set of integers was faithfully represented by Python. We cannot say the same for real numbers. Python, and all computing systems, have an approximation to the real numbers called "floating point numbers". (There is an alternative approximation called "fixed point numbers" but most programming languages, including Python, don't implement that so we won't bother with it.)

As you may know, $\mathbb{R}$ is the mathematical symbol for the real numbers. However, since the computer's floating-point numbers are only an approximation of the real numbers – and not a very good approximation at that – the authors are using a *crossed out* $\mathbb{R}$ to represent them in this course.

Again, those of you who have worked with numerical code in other programming languages will already be familiar with the vagaries of floating point arithmetic – treat this as a brief revision of the relevant concepts.

Those of you who are *not* familiar with the vagaries of floating point arithmetic should have a look at the article "The Perils of Floating Point" by Bruce M. Bush, available on-line at:

> http://www.lahey.com/float.htm

Note that all the examples in this article are in Fortran, but everything the article discusses is as relevant to Python as it is to Fortran.

```
>>> 1.0
1.0
```
Floating point number

```
>>> 0.5
0.5
```
½ is OK

Powers of two

```
>>> 0.25
0.25
```
¼ is OK

```
>>> 0.1
0.10000000000000001
```
1/10 is *not*

Usual issues with representation in base 2

We represent floating point numbers by including a decimal point in the notation. "1.0" is the floating point number "one point zero" and is quite different from the integer "1". (We can specify this to Python as "`1.`" instead of "`1.0`" if we wish.)

Even with simple numbers like this, though, there is a catch. We use "base ten" numbers but computers work internally in base two. The floating point system can cope with moderate integer values like 1·0, 2·0 and so on, but has a harder time with simple fractions. Fractions that are powers of two (half, quarter, eighth, etc.) can all be handled exactly correctly. Fractions that aren't, like a tenth for example, are approximated internally. We see a tenth (0·1) as simpler than a third (0·333333333…) only because we write in base ten. In base two a tenth is the infinitely repeating fraction 0·00011001100110011… Since the computer can only store a finite number of digits, numbers such as a tenth can only be stored approximately. So whereas in base ten, we can exactly represent fractions such as a half, a fifth, a tenth and so on, with computers it's only fractions like a half, a quarter, an eighth, etc. that have the privileged status of being represented exactly.

We're going to ignore this issue in this introductory course and will pretend that numbers are stored internally the same way we see them as a user.

Python provides a number of functions that perform various mathematical operations (such as finding the positive square root of a positive number) in one of its libraries – the `math` module (Python calls its libraries "*modules*"; we'll meet modules a little later). You can find out what functions are in this module by typing "`help('math')`" at the Python prompt, or from the following URL:

> `http://docs.python.org/library/math.html`

Most of the functions in the `math` module can be used on either integers or floating point numbers. In general, these functions will give their result as a floating point number even if you give them an integer as input.

17

```
>>> 2.0*2.0
4.0
>>> 4.0*4.0
16.0

…

>>> 65536.0*65536.0
4294967296.0
>>> 4294967296.0*4294967296.0
1.8446744073709552 e+19

17 significant figures
```

If we repeat the successive squaring trick that we applied to the integers everything seems fine up to just over 4 billion.

If we square it again we get an unexpected result. The answer is printed as

```
1.8446744073709552e+19
```

This means $1{\cdot}8446744073709552 \times 10^{19}$, which isn't the right answer.

In Python, floating point numbers are stored to only 17 significant figures of accuracy. Positive floating point numbers can be thought of as a number between 1 and 10 multiplied by a power of 10 where the number between 1 and 10 is stored to 17 significant figures of precision. (Recall that actually the number is stored in base 2 with a power of 2 rather than a power of 10. For our purposes this detail won't matter.)

In practice this should not matter to scientists. If you are relying on the sixteenth or seventeenth decimal place for your results you're doing it wrong!

What it *does* mean is that if you are doing mathematics with values that you think ought to be integers you should stick to the integer type, not the floating point numbers.

(Note for pedants: Python floating point numbers are really C `doubles`. This means that the number of significant figures of accuracy to which Python stores floating point numbers depends on the precision of the `double` type of the underlying C compiler that was used to compile the Python interpreter. On most modern PCs this means that you will get at least 17 significant figures of accuracy, but the exact precision may vary. Python does not provide any easy way for the user to find out the exact range and precision of floating point values on their machine.)

```
>>> 4294967296.0*4294967296.0
1.8446744073709552e+19

>>> 1.8446744073709552e+19*1.8446744073709552e+19
3.4028236692093846e+38

>>> 3.4028236692093846e+38*3.4028236692093846e+38
1.157920892373162e+77

>>> 1.157920892373162e+77*1.157920892373162e+77
1.3407807929942597e+154

>>> 1.3407807929942597e+154*1.3407807929942597e+154
inf
```

overflow

Limit at 2**1023

19

Just as there is a limit of 17 significant figures on the precision of the number there is a limit on how large the power of 10 can get. (On most modern PCs, the limit will be about 2^{1023}, although the exact limit depends on the hardware, operating system, etc.)   If we continue with the squaring just four more times after 4294967296.0 we get a floating point number whose exponent is too great to be stored. Python indicates this by printing "inf" as the answer. We have reached "floating point overflow", or "infinity".

Note that sometimes Python will give you an OverflowError (which will normally cause your script to stop executing and exit with an error) if the result of a calculation is too big, rather than an inf (which allows your script to carry on, albeit with "inf" as the answer instead of an actual number).

Machine epsilon

```
>>> 1.0 + 1.0e-16
1.0
>>> 1.0 + 2.0e-16
1.0000000000000002
```

too small to make a difference

large enough

```
>>> 1.0 + 1.1e-16
1.0
>>> 1.0 + 1.9e-16
1.0000000000000002
```

Spend the next few minutes using Python interactively to estimate machine epsilon – we'll write a Python program to do this for us a little later

The limit of 17 significant figures alluded to earlier begs a question. What is the smallest positive floating point number that can be added to $1{\cdot}0$ to give a number larger than $1{\cdot}0$? This quantity, known as *machine epsilon*, gives an idea of how precise the system is.

I'd like you to spend the next few minutes using Python interactively to estimate machine epsilon. This will give you an opportunity to familiarise yourself with the Python interpreter. Later today we will write a Python program to calculate an estimate of machine epsilon.

If you have not met the concept of machine epsilon before, you might like to take a look at the Wikipedia entry for machine epsilon for some references and more in-depth information:

http://en.wikipedia.org/wiki/Machine_epsilon

Strings

'Hello, world!'   '''Hello, world!'''

"Hello, world!"   """Hello, world!"""

We shall now briefly look at how Python stores text.

Python stores text as "strings of characters", referred to as "strings".

Single quotes	Double quotes
'Hello, world!**'**	**"**Hello, world!**"**
Single quotes around the string	Double quotes around the string

Exactly equivalent

22

Simple text can be represented as that text surrounded by either single quotes or double quotes. Again, because of the historical nature of keyboards, computing tends not to distinguish opening and closing quotes. The same single quote character, **'**, is used for the start of the string as for the end and the same double quote character, **"**, is used for the start and end of a string. However, a string that starts with a single quote must end with a single quote and a string that starts with a double quote must end with a double quote.

'He said "Hello, world!" to her.'

>>> **print 'He said "Hello, world!" to her.'**
He said "Hello, world!" to her.

"He said 'Hello, world!' to her."

>>> **print "He said 'Hello, world!' to her."**
He said 'Hello, world!' to her.

The advantage offered by this flexibility is that if the text you need to represent contains double quotes then you can use single quotes to delimit it. If you have to represent a string with single quotes in it you can delimit it with double quotes.

(If you have to have both then stay tuned.)

String concatenation

Two separate strings

```
>>> 'He said' 'something to her.'
'He saidsomething to her.'
```

Optional space(s)

```
>>> 'He said''something to her.'
'He saidsomething to her.'
```

```
>>> 'He said' + 'something to her.'
'He saidsomething to her.'
```

Can also use + operator

24

As you probably know, joining strings together is called "concatenation". (We say we have "concatenated" the strings.)

In Python we can either do this using the addition operator "+", or by just typing one string immediately followed by another (optionally separated by spaces). Note that no spaces are inserted between the strings to be joined, Python just joins them together exactly as we typed them, in the order we gave them to it. Note also that the original strings are *not* modified, rather Python gives us a *new* string that consists of the original strings concatenated.

Special characters

\n ➡ ↵   \' ➡ '

\t ➡ →| \" ➡ "

\a ➡ ♫   >>> **print 'Hello, \n world!'**
                         Hello,
\\ ➡ \   world!

                         "\n" converted
                         to "new line"

Within the string, we can write certain special characters with special codes. For example, the "end of line" or "new line" character can be represented as "\n". Similarly the instruction to jump to the next tab stop is represented with an embedded "tab" character which is given by "\t". Two other useful characters are the "beep" or "alarm" which is given by "\a" and the backslash character itself, "\" which is given by "\\".

This can also be used to embed single quotes and double quotes in a string without interfering with the quotes closing the string. Strictly speaking these aren't "special" characters but occasionally they have to be treated specially.

Long strings

"""Long pieces of
text are easier to
handle if literal new
lines can be
embedded in them.**"""**

26

There's one last trick Python has to offer with strings.

Very long strings which cover several lines need to have multiple "\n" escapes within them. This can prove to be extremely awkward as the editor has to track these as the string is edited and recast. To assist with this, Python allows the use of triple sets of double or single quotes to enclose a string, within which new lines are accepted literally.

Single quotes and individual (i.e. not triple) double quotes can be used literally inside triple double quoted strings too.
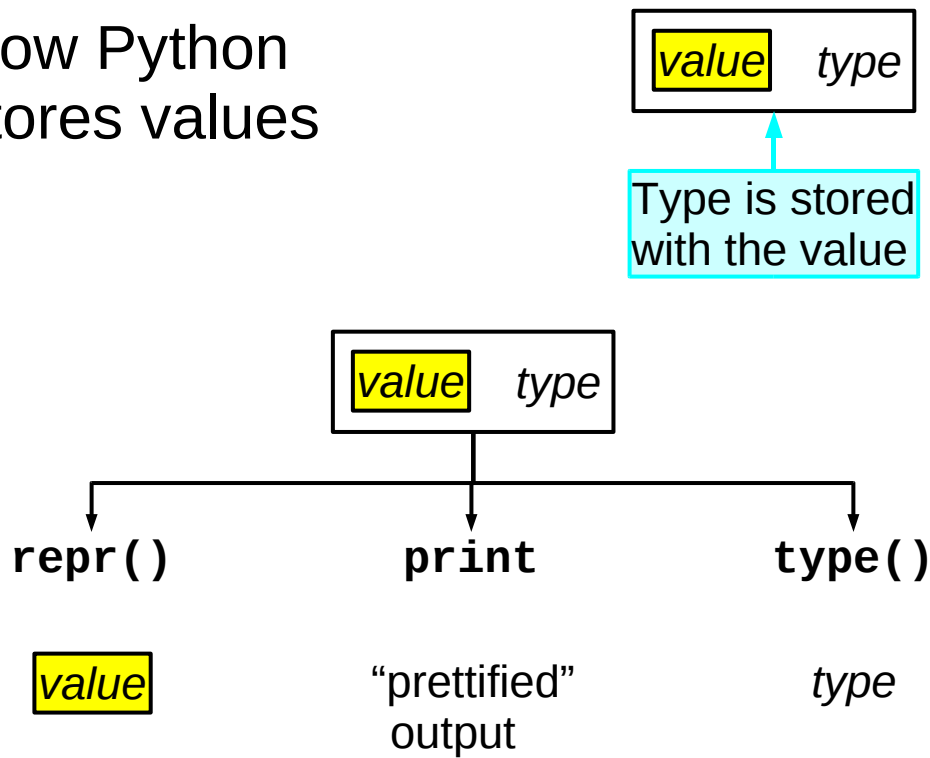
Long strings

Triple single quotes

'''Long pieces of
text are easier to
handle if literal new
lines can be
embedded in them.'''

27

Triple double quotes are more common but triple single quotes also work.

How Python stores values

value *type*

Type is stored with the value

value *type*

repr()   **print**   **type()**

value   "prettified" output   *type*

28

We will take a brief look at how Python stores its values because this is one of the features that distinguishes languages from each other.

Python is a *dynamically typed* language, which means that it decides on the type of a variable (we'll start actually using variables soon) when you assign a value to the variable (technically, it decides on the variable's type at "run-time").

Python is also a *strongly typed* language, which means that once something has been given a type, you can't change that type. (You can, of course, re-define a variable so that it has a different type, but you can't change the type without re-defining it.)

In Python (and most other interpreted languages), a record is kept of what type of value it is (integer, floating point number, string of characters, etc.) alongside the system memory used to store the actual value itself. This means that the type of a variable is determined by the type of the value which that variable contains.

There are three ways of getting at Python's stored values.

The `print` command which we already met outputs a "prettified" version of the value. We will see this prettifying in a moment. We can also instruct Python to give us the type of the value with a function called "`type()`". Finally, we can tell Python to give us the value in a raw, unprettified version of the value with a function called "`repr()`" (because it gives the internal **repr**esentation of the value). The `repr()` function displays the raw version of the value as a *string* (hence when you invoke the `repr()` function the result is displayed surrounded by single quotes). (Note that in certain circumstances the `repr()` function may still do some "prettifying" of values in order to display them as reasonable strings, but usually much less so than `print` does.)

We include parentheses (round brackets) after the names of the "`repr()`" and "`type()`" functions to indicate that, in common with most functions, they require parentheses (round brackets) around their arguments. It's `print` that is the special case here.

```
>>> print 1.2345678901234567
1.23456789012

>>> type( 1.2345678901234567 )
<type 'float'>

>>> repr( 1.2345678901234567 )
'1.2345678901234567'
```

For many types there's no difference between the `print`ed output and the `repr()` output. For floating point numbers, though, we can distinguish them because `print` outputs fewer significant figures.

The easiest way to see this is to look at the float 1·2345678901234567. If we `print` it we get output looking like 1·23456789012, but if we apply the `repr()` function we see the full value.

The `type()` function will give the type of a value in a rather weird form. The reasons are a bit technical but what we need to know is that `<type 'float'>` means that this is a floating point number (called a "`float`" in the jargon).

Again, note that `repr()` and `type()`, in common with most functions, require parentheses (round brackets) around their arguments. It's `print` that is the special case here.

Two other useful types

Complex
```
>>>  (1.0 + 2.0j) * (1.5 + 2.5j)
(-3.5+5.5j)
```

Boolean
```
>>>  True and False
False

>>>  123  ==  234
False
```

There are two other useful types we need to know about:

- **Complex numbers:** A pair of floats can be bound together to form a complex number (subject to all the caveats applied to floats representing reals). The complex numbers are built into Python, but with the letter "j" representing the square root of -1, rather than *i*.

  The complex numbers are our first example of a "composite type", built up out of other, simpler types.

  For more details on some of the operations available to you if you are using complex numbers in Python see the "Numeric Types" sub-section of *The Python Standard Library* reference manual:

  http://docs.python.org/library/stdtypes.html#numeric-types-int-float-long-complex

  Python also provides complex number versions of many of the functions in the `math` module in another module, the `cmath` module – for details, type "**help('cmath')**" at the Python prompt or see:

  http://docs.python.org/library/cmath.html

- **Booleans:** In Python, the truth or falsehood of a statement is a value. These values are of a type that can only take two values: `True` and `False` (with the obvious meanings). This type is called a "Boolean", named after George Boole, a mathematician who developed the algebra of these sorts of values, and is called a "`bool`" in Python. Zero (whether the integer zero (`0`), floating point zero (`0.0`) or complex zero (`0.0 + 0.0j`)) and the empty string are equivalent to `False`, all other numbers and strings are equivalent to `True`.

  Booleans arise as the result of tests, indicating whether the test is passed or failed. For example a Boolean would be returned if we tested to see if two numbers were the same (which we do with "`==`", as on the slide above). `True` would be returned if they were, and `False` would be returned if they weren't. And, as we might expect, Booleans can also be combined with `and` or `or`.

Comparisons

```
>>> 1 == 2          >>> 'abc' == 'ABC'
False               False

>>> 1 < 2           >>> 'abc' < 'ABC'
True                False

>>> 1 >= 2          >>> 'abc' >= 'ABC'
False               True

>>> 1 == 1.0
True
```

This brings us on to the subject of tests. The Python test for two things being equal is a double equals sign, "==" (note that there are *no* spaces between the two equal signs). A single equals sign means something else, which we will meet later.

There are also various other comparison tests, such as "greater than" (>), "less than" (<), "greater than or equal to" (>=), and "less than or equal to" (<=). You will note that "not equal to" is omitted from this list. We will return to that particular test in a moment.

You will note that string comparisons are done case sensitively, so that the string 'a' is not the same as the string 'A'. Strings are compared lexicographically with the lower case letters being greater than the upper case letters. (Python compares the strings lexicographically using their numeric representation as given by the ord() function. For example, ord('a') = 97 and ord('A') = 65, hence 'a' > 'A'.)

Note also that Python will automatically convert numbers from one type to another (this is known as "*coercion*") for the purposes of a comparison test. Thus the integer 1 is considered to be equal to the floating point number 1.0.

… not equal to …

```
>>>  not 1 == 2        >>>  not 'abc' == 'ABC'
True                   True




>>>  1 != 2            >>>  'abc' !=  'ABC'
True                   True
```

There are two ways to express "is not equal to". Because this is a common test, it has its own operator, equivalent to "==", ">=" etc. This operator is "!=". However, Python has a more general "is not" operator called, logically enough, "not". This can precede any expression that evaluates to a Boolean and inverts it; True becomes False and *vice versa*.

Conjunctions

```
>>>  1 == 2 and 3 == 3
False

>>>  1 == 2 or 3 == 3
True
```

As mentioned earlier, you can join Booleans with the conjunctions and or or. As you might expect, you can also join tests with these conjunctions.

The "and" logical operator requires both sub-tests to be True, so its "truth table" is as shown below:

$Test_1$	$Test_2$	$Test_1$ and $Test_2$
True	True	True
True	False	False
False	True	False
False	False	False

The "or" operator requires only one of them to be True (although if they are both True, that's fine as well), so its "truth table" is:

$Test_1$	$Test_2$	$Test_1$ or $Test_2$
True	True	True
True	False	True
False	True	True
False	False	False

Python does so-called "short-circuit evaluation" or "minimal evaluation" – (sometimes (incorrectly) called "lazy evaluation") – when you combine tests: it will evaluate as few tests as it can possibly manage to still get the final result. So, when you combine two tests with "and", if the result of the first test is False, then Python won't even bother evaluating the second test, since both "False and False" and "False and True" evaluate to False. Consequently, if the first test evaluates to False, the result of combining the two tests with "and" will also be False, regardless of whether the second test evaluates to True or False. Similarly, when you combine two tests with "or", if the result of the first test is True, then Python won't even bother evaluating the second test, since both "True or False" and "True or True" evaluate to True.

Evaluate the following Python expressions in your head:

```
>>>  2 - 2 == 1 / 2
>>>  True and False or True
>>>  1 + 1.0e-16 > 1
>>>  5 == 6 or 2 * 8 == 16
>>>  7 == 7 / 2 * 2
>>>  'AbC' > 'ABC'
```

Now try them interactively in Python and see if you were correct.

Here's another chance to play with the Python interpreter for yourself.

I want you to evaluate the following expressions in your head, i.e., for each of them decide whether Python would evaluate them as True or as False. Once you have decided how Python would evaluate them, type them into the Python interpreter and see if you were correct.

If you have any problems with this exercise, or if any questions arise as a result of the exercise, please ask the course giver or a demonstrator.

Give yourself 5 minutes or so to do this exercise (or as much as you can manage in that time) and then take a break. Taking regular breaks is very important when using the computer for any length of time. (Note that "take a break" means "take a break from this computer" *not* "take a break from this course to check your e-mail".)

In case you are wondering in what order Python evaluates things, it uses similar rules for determining the order in which things are evaluated as most other programming languages. We'll look briefly at this order after the break.

Precedence

First	x**y	
	-6, +6	Arithmetic operations
	x/y, x*y, x%y	
	x+y, x-y	
	x<y, x<=y, …	
	x **in** y, x **not in** y	
	not x	Logical operations
	x **and** y	
Last	x **or** y	

35

Having now got Python to evaluate various expressions, you may be wondering about the order in which Python evaluates operations within an expression. Python uses similar rules for *precedence* (i.e. the order in which things are evaluated) as most other programming languages. A summary of this order is shown on the slide above.

Python expressions are evaluated from left to right, but with operators being evaluated in order of precedence (highest to lowest). First any arithmetic operations are evaluated in order of precedence, then any comparisons (<, ==, etc.) and then any logical operations (such as `not`, `and`, etc.).

(Note that we have not yet met all of the operations shown on the slide above.)

You can also find a summary of the precedence order of all the operators in Python in the Python documentation, although note that the documentation lists the operators in ascending order of precedence (i.e. from lowest precedence to highest precedence, rather than from highest to lowest as we have done here):

http://docs.python.org/reference/expressions.html#evaluation-order

In common with most other programming languages you can change the order in which parts of an expression are evaluated by surrounding them with parentheses (round brackets), e.g.

```
>>> 2 + 3 * 5
17
>>> (2 + 3) * 5
25
```

Flow control in Python: `if`

```
if x > 0.0 :
        print 'Positive'

elif x < 0.0 :
        print 'Negative'
        x = -1.0 * x

else :
        print 'Zero'
```

compulsory

indentation

optional, repeatable

multiple lines indented

optional

Python has several constructs for controlling the flow of your program. The first one we will look at is the `if` statement.

As one might expect, the `if` statement specifies a test and an action (or series of actions) to be taken if the test evaluates to `True`. (We will refer to these actions as the "`if`" block.)

You can then have as many (or as few) "else if" clauses as you wish. Each "else if" clause specifies a further test and one or more actions to carry out if its test evaluates to `True`. These "else if" clauses are introduced by the keyword "`elif`". (We will refer to the actions specified in an "else if" clause as an "`elif`" block.)
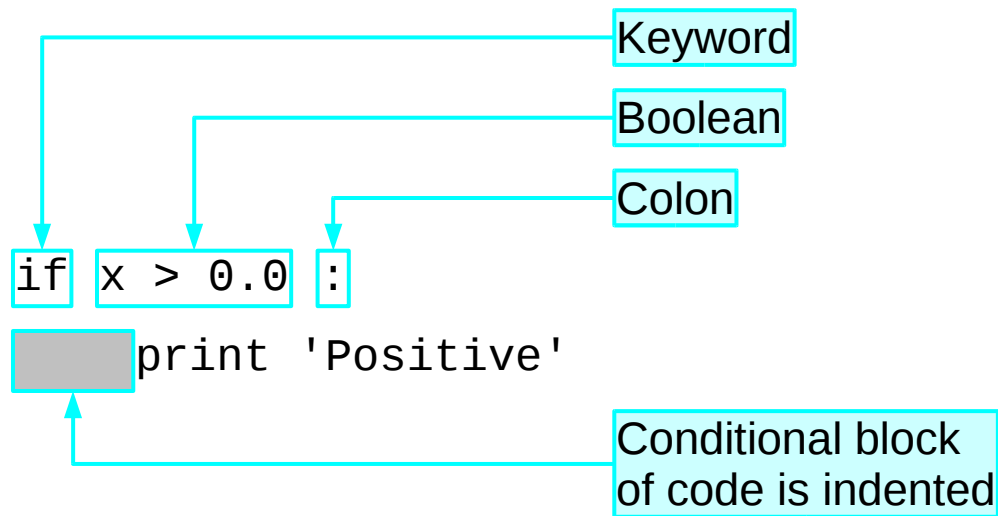
In order for an "else if" clause to be evaluated, the test specified by `if` must evaluate to `False`, *and* all the tests of any preceding "else if" clauses that are part of the `if` statement must also evaluate to `False`.

Finally, you can optionally have an "else" clause that specifies an action to be taken if *all* preceding tests in the `if` statement (including any from any "else if" clauses) evaluated to `False`. This "else" clause is introduced by the keyword "`else`" immediately followed by a colon (`:`). We will refer to the actions specified in this clause as the "`else`" block.

You will note that each of the lines starting "`if`", "`elif`" or "`else`" end in a colon (:). This is standard Python syntax that means "I've finished with this line, and all the following indented lines belong to this section of code". Python uses *indentation* to group related lines of code. Whenever you see a colon (`:`) in a Python script, the next line **must** be indented.

The amount of indentation doesn't matter, but all the lines in a block of code must be indented by the same amount (except for lines in any sub-blocks, which will be further indented). If this seems a bit strange compare it with official documents with paragraphs, sub-paragraphs and sub-sub-paragraphs, each of which is indented more and more.

Flow control in Python: **if**

```
        Keyword

        Boolean

        Colon

if  x > 0.0  :
     print 'Positive'

        Conditional block
        of code is indented
```

Since this is the first flow control construct we've met in Python let's examine it in a bit more closely. This will also give us a chance to get to grips with Python's use of indentation.

The first line of an `if` statement is of the form "`if test:`".

The test is preceded with the Python keyword "`if`". This introduces the test and tells the Python interpreter that a block of code is going to be executed or not executed according to the evaluation of this test. We shall refer to this block of code as the "`if`" block.

The test itself is followed by a colon (`:`). This is standard Python to indicate that the test is over and the conditionally executed block is about to start. It ends the line.

The "`if test:`" line is followed by the conditional block of Python. This is indented (typically by a small number of space characters or a tab stop). Every line in that block is indented by the same amount (unless it contains sub blocks of its own which are indented further relative to it). Python uses indentation to mark the block. As soon as the indentation has stopped the "`if`" block ends and a new code block starts.

Nested indentation

```
if x > 0.0 :
        print 'Positive'
else :
    if x < 0.0 :
            print 'Negative'
            x = -1.0 * x
    else :
            print 'Zero'
```

As mentioned earlier, if we have any sub-blocks, they will be further indented, as seen in the example above.

In the above example, in our "else" block we have a new if statement. The conditional blocks of this if statement will be further indented as they are sub-blocks of the "else" block.

Flow control in Python: **while**

```
while x % 2 == 0 :
        print x, 'still even'
        x = x/2


else :
        print x, 'is odd'
```

compulsory

optional

The next flow control construct we will look at is a loop, the `while` loop, that does something as long as some test evaluates to `True`.

The idea behind this looping structure is for the script to repeat an action (or series of actions) as long as the specified test evaluates to `True`. Each time the script completes the action (or series of actions) it evaluates the test again. If the result is `True`, it repeats the action(s); if the result is `False` it stops. (We will refer to the action (or series of actions) as the "`while`" block.)

The syntax for the `while` loop is "`while test:`".

This is immediately followed by the block of Python (which can be one or more lines of Python) to be run repeatedly until the test evaluates to `False` (the "`while`" block).

There is also an optional "else" clause which, if present, is executed when the test specified in the `while` loop evaluates to `False`, i.e. at the end of the `while` loop. In practice, this "else" clause is seldom used. As with the `if` statement, the "else" clause is introduced by "`else:`" followed by an indented "`else`" block.

(The reason that you might want to use the "`else`" block is that if your program "breaks out" of a `while` loop with the `break` statement then the "`else`" block will be skipped. For this introductory course, we're not going to bother with the `break` statement.)

```
#!/usr/bin/python


epsilon = 1.0



while 1.0 + epsilon > 1.0:
    epsilon = epsilon / 2.0



epsilon = 2.0 * epsilon


print epsilon
```
epsilon.py

Approximate
machine
epsilon

$1.0 + \varepsilon\ > 1.0$
$1.0 + \varepsilon/2 == 1.0$

To illustrate the `while` loop we will consider a simple script to give us an approximation to machine epsilon. Rather than trying to find the smallest floating point number that can be added to 1·0 without changing it, we will find a number which is large enough but which, if halved, isn't. Essentially we will get an estimate between ε and 2ε.

We start (before any looping) by setting an approximation to the machine epsilon that's way too large. We'll take 1.0 as our too large initial estimate:

```
epsilon = 1.0
```

We want to keep dividing by 2.0 so long as our estimate is too large. What we mean by too large is that adding it to 1.0 gives us a number strictly larger than 1.0. So that's the test for our `while` loop:

```
1.0 + epsilon > 1.0
```

What do we want to do each time the test is passed? We want to decrease our estimate by halving it. So that's the body of the `while` loop:

```
epsilon = epsilon / 2.0
```

Once the test fails, what do we do? Well, we now have the estimate one halving too small so we double it up again. Once we have done this we had better print it out:

```
epsilon = epsilon * 2.0
print epsilon
```

And that's it!

You can find this script in your course home directories as `espilon.py`.

PS: It works.

If you run this script on different machines you may get different answers. Machine epsilon is machine specific, which is why it is called *machine* epsilon and not "floating point epsilon".

```python
#!/usr/bin/python

# Start with too big a value
epsilon = 1.0

# Halve it until it gets too small
while 1.0 + epsilon > 1.0:
    epsilon = epsilon / 2.0

# It's one step too small now,
# so double it again.
epsilon = 2.0 * epsilon

# And output the result
print epsilon
```

epsilon.py

As I'm sure you all know, it is very important that you comment your code while you are writing it. If you don't, how will you remember what is does (and why) next week? Next month? Next *year*?

So, as you might expect, Python has the concept of a "comment" in the code. This is a line which has no effect on what Python actually does but just sits there to remind the reader of what the code is for.

Comments in Python are lines whose first non-whitespace character is the hash symbol "#". Everything from the "#" to the end of the line is ignored.

(You can also put a comment at the end of a line: in this case, anything that follows the hash symbol (#) until the end of the line will be ignored by Python.)

Time for a break…

Have a look at the script `epsilon2.py` in your home directory.

This script gives a better estimate of machine than the script we just wrote.

See if you can figure out what it does – if there is anything you don't understand, tell the course giver or a demonstrator.

Examine the script `epsilon2.py` in your course home directories to see a superior script for approximating machine epsilon. This uses nothing that we haven't seen already but is slightly more involved. If you can understand this script then you're doing fine.

You should also run the script and test its output.

And then it's time for a break. As you should all know, you need to look after yourselves properly when you're using a computer. It's important to get away from the computer regularly.

You need to let your arms and wrists relax. Repetitive strain injury (RSI) can cripple you.

Your eyes need to have a chance to relax instead of focussing on a screen. Also while you are staring intently at a screen your blink rate drops and your eyes dry out. You need to let them get back to normal.

You also need to let your back straighten to avoid long term postural difficulties.

Computing screws you up, basically.

```
#!/usr/bin/python

too_large = 1.0
too_small = 0.0
tolerance = 1.0e-27

while  too_large - too_small  >  tolerance:

   mid_point = (too_large + too_small)/2.0

   if 1.0 + mid_point > 1.0:
      too_large = mid_point
   else:
      too_small = mid_point

print  too_small,  '< epsilon <',  too_large
```

A better estimate for machine epsilon

epsilon2.py

This is the `epsilon2.py` script you were asked to look at before the break.

There are two quite distinct aspects to understanding it: *what* the Python does, and *why* it does it.

In terms of what it does, this is a relatively simple script. It sets up three values at the start and then runs a `while` loop which modifies one of those values at a time until some condition is broken. Inside the `while` loop a new value is calculated and then an `if` statement is run depending on some other test. One value or another is changed according to the results of that test. Once the `while` loop is done it prints out three values on a line. The Python contains an `if` statement nested inside a `while` loop but other than that it is quite straightforward.

It's what is does that's the clever bit. As many of you will know, the posh, computer-y name for "what it does" is *algorithm*.

The algorithm here uses a technique called *bisection*. We start with a range of real numbers which we know must contain machine epsilon. Then we calculate the mid-point of that range and ask if that number is bigger or smaller than the machine epsilon. If the estimate is too small we can change the lower bound of our interval to the value of this mid-point. If it is too large we can change the upper bound of our interval to this mid-point estimate. Either way we get an interval that contains the real value that is half the length of the interval we started with (i.e., the interval has been *bisected*). This is the `if` statement of our Python. We can repeat this to halve the length of our interval time and time again until the length of the interval is short enough for us. This is the `while` statement in our Python script.

Some of you may be wondering why we keep going until the length of our interval is less than 10^{-27}. Where does the value of 10^{-27} come from? This value was chosen because, on the PCs we are using for this course, if the interval is any smaller than this then when we use `print` to display its lower and upper bounds there will be no difference on the screen between the two values displayed. This is because `print` will not display these bounds to enough significant figures for us to see that they are different from each other if the difference between them is less than 10^{-27}. (Recall that `print` may "prettify" values; in particular, for floating point numbers it only displays a certain number of significant figures. Note that, as machine epsilon is *machine specific,* on other computers you may need to use a higher or lower value than 10^{-27}.)

Lists

H He Li Be B C N O F Ne Na Mg Al Si P S Cl Ar

Red Orange Yellow Blue Indigo Violet

44

We've already met various simple data types in Python. We've also seen that Python has "composite" types, built up of the simpler data types, for example, complex numbers are such a composite type (built up of two floating point numbers).

Python also has a more general type designed to cope with arbitrary lists of values and it is this type and a corresponding control structure that we will now discuss.

The Python data type for storing a collection of sequential, related data is the "`list`". Unlike a complex number − which always contains two floating point numbers − a list can contain any number of items, and the items can be of any type. The items in a list don't all have to be of the same type, although it is generally a bad idea to have lists whose items are of different types. (If you need to do this, you should use a different structure, known as a "`tuple`", instead. We'll be meeting tuples later.)

Python's lists are roughly equivalent to most other languages' arrays, except that most languages require their arrays to be of fixed length and all the items in the array to be of the same type.

(The individual items in the list (or indeed in any type of sequence) are often referred to as the *elements* of the list (or sequence), although we will try to avoid using this terminology, as it can be confusing in a scientific context, where "element" usually means something different, e.g. the atomic elements given in the periodic table.)

```
>>> [ 2, 3, 5, 7, 11, 13, 17, 19]
[ 2, 3, 5, 7, 11, 13, 17, 19]

>>> type([ 2, 3, 5, 7, 11, 13, 17, 19])
<type 'list'>

>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]
```

So how do we create lists in Python?

The syntax for representing a list is relatively straightforward. The items forming the list appear in order, are separated by commas and are surrounded by square brackets (`[]`). (The brackets must be square rather than any of the other sorts of brackets you may have at your disposal on your keyboard.)

Do note that this is not just syntactic sugar. A list is a genuine Python type, on a par with integers or floating point numbers.

As mentioned earlier, the type here is just "`list`" and not "list of integers". Some languages are more prescriptive about this sort of type; Python isn't.

When working with lists, it is very common to pick variable names for our lists that are the plural of whatever is in the list. So we might use `prime` as the name of the variable that corresponds to a single prime number from the list and we would use `primes` as the name of the variable corresponding to the list itself.

```
            0  1  2  3   4   5   6   7

>>>  primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ]

>>>  primes[2]
5

                    Indexing starts at 0
```

As with most languages that have this sort of data type, we start counting the items in the list from zero, not one. So what we would regard as "the third item in the list" is actually referred to by Python as "item number *two*". The number of the item in the list is known as its *index*.

The way to get an individual item from a list is to take the list, or the variable name corresponding to the list, and to follow it with the index of the item wanted in square brackets ([ ]), as shown in the example on the slide above. (Note that these are square brackets because that's what Python uses around indices, not square brackets because that's what lists use.)

```
          0  1  2  3  4  5  6  7

>>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19 ]

>>> primes[-1]
19        -8 -7 -6 -5 -4 -3 -2 -1
```

Perhaps slightly surprisingly, we can ask for the "minus first" item of a list. Python has a dirty trick that is occasionally useful where, if you specify a negative index, it counts from the end of the list.

This negative index trick can be applied all the way back in the list.

If a list has eight items (indexed 0 to 7) then the valid indices that can be asked for run from -8 to 7.

```
                          0  1  2  3  4  5  6  7
                          ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
        >>> primes = [ 2, 3, 5, 7, 11, 13, 17, 19]


        >>> primes[8]                              Where the
                                                   error was.
        Traceback (most recent call last):
          File "<stdin>", line 1, in <module>
        IndexError: list index out of range

                                                   The error
                                                   message.

                                                          48
```

So what happens if we ask for an invalid index? We get an error.

As this is our first error message we will take our time and inspect it carefully.

The first line declares that what you are seeing is a "traceback" with the "most recent call last". A traceback is the command's history: how you got to be here making this mistake, if you like. The error itself will come at the end. It will be preceded with how you got to be there. (If your script has jumped through several hoops to get there you will see a list of the hoops.) In our example we jumped straight to the error so the traceback is pretty trivial.

The next two lines are the traceback. In more complex examples there would be more than two but they would still have the general structure of a "file" line followed by "what happened" line.

The file line says that the error occurred in a file called "<stdin>". What this actually means is that the error occurred on Python being fed to the interpreter from "standard input". Standard input means your terminal. You typed the erroneous line and so the error came from you.

Each line at the ">>>" prompt is processed before the prompt comes back. Each line counts as "line 1".

If the error had come from a script you would have got the file name instead of "<stdin>" and the line number in the script.

The "<module>" refers to the module or function you were in. We haven't done modules or functions yet so you weren't in either a module or a function.

The third line gives information about the error itself. First comes a description of what type of error has happened. This is followed by a more detailed error message. If the error had come from a script the line of Python would be reproduced too.

Counting from zero and the **len()** function

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19]
>>> primes[0]
2
>>> primes[7]
19
>>> len(primes)
8
```

0 ≤ index ≤ 7

length 8

49

We can confirm that Python does count from zero for list indices by asking explicitly for item number zero.

The len() function returns the number of items in (i.e. the **len**gth of) a list.

Changing an item in a list

```
>>> data = [56.0, 49.5, 32.0]
>>> data[1]
49.5
>>> data[1] = 42.25
>>> data
[56.0, 42.25, 32.0]
```

"item number 1" ("2nd item")

Assign new value to "item number 1" in list

List is modified "in place"

50

As you would expect, we can easily change the value of a particular item in a list provided we know its index. The syntax is:

```
listname[index] = new_value
```

Note that this changes the list itself (i.e. the list is modified in place), it does not create a new list with the new value and then make the old list equal to this new list.

Note also that this is **not** a way to add a new item to the end of a list. (We will see how to do that in a little while.)

Empty lists

>>> **empty = [ ]**

>>> **len(empty)**
0

>>> **len([ ])**
0

Note that it is quite possible to have an empty list. The list `[]` is perfectly valid in Python.

A common trick for creating a list is to start with an empty list and then to append items to it one at a time as the program proceeds.

(Note that in the slide above we could have used any variable name for our empty list; we didn't have to call it `empty`. There's nothing special about the word "empty" in Python.)

Single item lists

A list with one item is not the same as the item itself!

```
>>> [1234] == 1234          >>> type([1234])
False                       <type 'list'>

                            >>> type(1234)
                            <type 'int'>
```
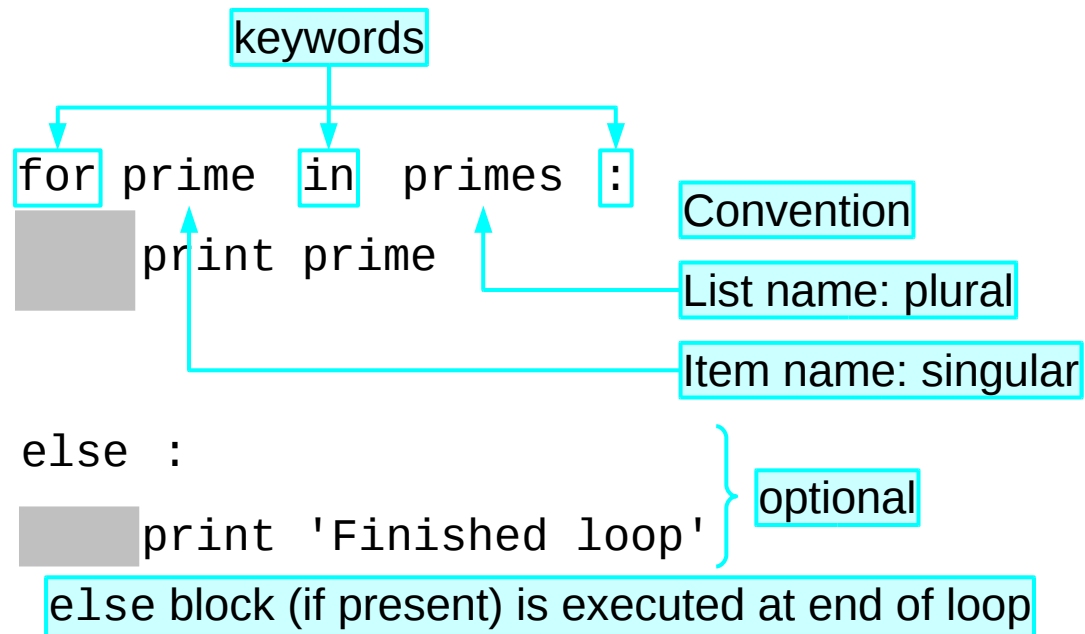
It is also possible to have a list which only has a single item.

Please note that a list with one item in it is quite different from the item itself. Some languages deliberately confuse the two. While it may make some simple tricks simpler it makes many, many more things vastly more complicated.

Flow control in Python: **for**

keywords

```
for prime  in  primes :
          print prime
```

Convention

List name: plural

Item name: singular

```
else :
          print 'Finished loop'
```
} optional

else block (if present) is executed at end of loop

There is a special loop construct, the `for` loop, that does something for every item in a list.

The idea behind this looping structure is for the script to get through the list, one item at a time, asking if there are any items left to go. If there are a variable (known as the "loop variable") gets set to refer to the current item in the list and a block of code is run. Then it asks whether there are any more items left to go and, if so, runs the same block of code, and so on until there are no more items left in the list.

The syntax for the `for` loop is very similar to the `while` loop but instead of "`while test:`" we have "`for item in list:`".

The words "`for`" and "`in`" are just syntax. The word immediately after "`for`" is the name of the "loop variable" − the variable that is going to be used to track the items in the list that appears immediately after "`in`".

The block of Python (typically several lines of Python rather than just one) to be run for every item in the list is indented just as it was for the `while` loop.

There is also an optional "`else`" block which, if present, is executed at the end of the `for` loop, i.e. after it has finished processing the items in the list. In practice, this "`else`" block is almost never used. As with the `while` loop, the "else" clause is introduced by "`else:`" followed by an indented "`else`" block.

(The reason that you might want to use the "`else`" block is that if your program "breaks out" of a `for` loop with the `break` statement then the "`else`" block will be skipped. For this introductory course, we're not going to bother with the `break` statement.)

Warning: loop variable persists

Definition of loop variable

```
for prime  in  primes :
        print prime
print 'Done!'
print prime
```

Correct use of loop variable

Improper use of loop variable

But legal!

And now a warning.

The loop variable, `prime` in the example above, persists after the loop is finished, which means we can reuse it. This is not a good idea, as in long scripts we can easily confuse ourselves if we have variables whose names are the same as any of our loop variables. We created the loop variable for use in our `for` loop, and we really shouldn't use it anywhere else.

Loop variable "hygiene"

Create loop variable

```
for prime  in  primes :
        print prime
del prime
print 'Done!'
```

Use loop variable

Delete loop variable

This brings us to a Python operator which we will use to encourage code cleanliness: `del`. The `del` operator deletes a variable from the set known about by the interpreter.

The well-written `for` loop has a self-contained property about itself.

Before the loop starts the interpreter knows about the list and not the loop variable. The "`for`" line defines an extra variable, the loop variable, which is then used in the loop. Using the `del` operator, we can delete this loop variable on completion of the `for` loop. So now after the loop the interpreter is exactly the same state as it was in before; it knows about the list but not the loop variable. The loop has left the system "unscathed".

```
#!/usr/bin/python

# This is a list of numbers we want
#  to add up.
weights = [ 0.1, 0.5, 2.6, 7.0, 5.3 ]

# Add all the numbers in the list
#  together.

                    What goes here?


# Print the result.
print
                                          addition.py
```

And now for an exercise.

In your course home directories you will find an incomplete script called `addition.py`.

Complete the script so that it adds up all the numbers in the list `weights` and prints out the total. Obviously, one way of doing this would be to manually type the numbers in the list out as a long sum for Python to do. That would not be a very sensible way of doing things, and you wouldn't learn very much by doing it that way. I suggest you try something else.

If you have any questions about anything we've covered so far, now would be a good time to ask them.

```python
#!/usr/bin/python

# This is a list of numbers we want
#  to add up.
weights = [ 0.1, 0.5, 2.6, 7.0, 5.3 ]

# Add all the numbers in the list
#  together.
total = 0.0
for weight in weights:
    total = total + weight
del weight

# Print the result.
print total
```

addition.py

57

And above is a solution to the exercise.

…If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

Lists of anything

primes = [2, 3, 5, 7, 11, 13, 17, 19]

List of integers

names = ['Alice', 'Bob', 'Cathy', 'Dave']

List of strings

roots = [0.0, 1.57079632679, 3.14159265359]

List of floats

lists = [[1, 2, 3], [5], [9, 1]]

List of *lists*

We've mentioned already that the Python type is "list" rather than "list of integers", "list of floating point numbers", etc. It is possible in Python to have lists of anything (including other lists).
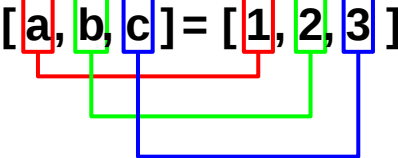
Mixed lists

stuff = [2, 'Bob', 3.14159265359, 'Dave']

Legal, but not a good idea.

See "tuples" later.

It is also possible to have lists with mixed content where the 1st item in the list is an integer, the 2nd a string, the 3rd a floating point number, etc. However, it is rarely a good idea. If you need to bunch together collections of various types then there is a better approach, called the "`tuple`", that we will meet later.

Lists of variables

```
>>>  [a, b, c] = [1, 2, 3]
```

```
>>> a
1
>>> b
2
>>> c
3
```

We can also have lists of variables, both to contain values and to have values assigned to them.

A list of variables can be assigned to from a list of values, so long as the number of items in the two lists is the same.

All or nothing

>>> **[d, e, f] = [1, 2, 3, 4]**

Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: too many values to unpack

Error message

>>> **d**

Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined

61

However, if there are too few variables (or too many values) no assignment is done.

```
>>> [ d, e, f ] = [ 1, 2, 3, 4 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined
>>> e
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'e' is not defined
>>> f
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f' is not defined
>>>
```

Note that we do not get d, e and f assigned with an error for the dangling 4. We get no assignment at all.

All or nothing

>>> **[g, h, i, j] = [1, 2, 3]**

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack

>>> **g**
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined

Error message

Similarly, if there are too many variables (or too few values) no assignment is done.

```
>>> [ g, h, i, j ] = [ 1, 2, 3 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
>>> g
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
>>>
```

Concatenating lists

Operator: "+"

```
>>>   ['H', 'He', 'Li'] + ['Be', 'B', 'C']
```

['H', 'He', 'Li', 'Be', 'B', 'C']

Python uses the "**+**" character for adding two lists together (adding two lists together is called "concatenating" the lists).

Please note that lists are not the same as sets. If an item appears in two lists being concatenated then the item appears twice in the new list.
```
>>> ['H', 'He', 'Li'] + ['Li', 'Be', 'B']
['H', 'He', 'Li', 'Li', 'Be', 'B']
```

Similarly, order matters. Concatenating list A with list B is not the same as concatenating B with A.
```
>>> ['H', 'He', 'Li'] + ['Be', 'B', 'C']
['H', 'He', 'Li', 'Be', 'B', 'C']
>>> ['Be', 'B', 'C'] + ['H', 'He', 'Li']
['Be', 'B', 'C', 'H', 'He', 'Li']
```

It's worth noting in passing that concatenating two lists takes the two lists and creates a third. It does not modify one list by adding the other list's items at the end.

Appending an item: **append()**

```
>>> symbols = [ 'H', 'He', 'Li', 'Be' ]
>>> symbols
[ 'H', 'He', 'Li', 'Be' ]

>>> symbols.append('B')

>>> symbols
[ 'H', 'He', 'Li', 'Be', 'B' ]
```

appending is a "method"

the item to append

no value returned

the list itself is changed

This may trigger a sense of déjà vu. We've added an item to the end of a list. Isn't this what we have already done with concatenation?

No it isn't, and it's important to understand the difference. Concatenation joined two *lists* together. Appending (which is what we are doing here) adds an *item* to a list. Recall that a list of one item and the item itself are two completely different things in Python.

Appending lets us see a very common Python mechanism so we will dwell on it for a moment. Note that the Python command is *not* "append(symbols, 'B')" or "append('B', symbols)" but rather it is "symbols.append('B')".
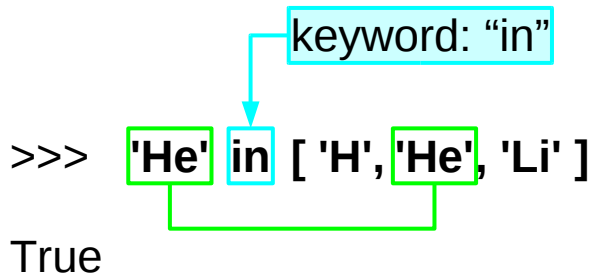
The append() function can only work on lists. So rather than tempt you to use it on non-lists by making it generally available it is built in to lists themselves (the technical term for this is *encapsulation*). Almost all Python objects have these sorts of built in functions (called "methods" in the jargon) but appending an item to a list is the first time we have encountered them.

A method is a function built in to a particular type so all items of that type will have that method. You, the programmer, don't have to do anything. The name of the method follows the thing itself separated by a dot. In all other ways it is exactly like a function that uses brackets, except that you don't need to put the object itself in as an argument.

This looks like a lot of fuss for no gain. In practice, this sort of organisation makes larger, more complex scripts vastly easier to manage. Of course, we're still at the stage of writing very simple scripts so we don't see that benefit immediately.

Note that what the append() method does is change the actual list itself, it does not create a new list made up of the original list with a new item added at the end. As append() changes the list itself, it does not return a value when we use it, but just silently updates the list "in place".

Membership of lists

keyword: "in"

```
>>>  'He' in [ 'H', 'He', 'Li' ]
True

>>>  'He' in [ 'Be', 'B', 'C' ]
False
```

We need a means to test to see if a value is present in a list.

The keyword "in", used away from a "for" statement, is used to test for presence in a list. The Python expression

        item in *list*

evaluates to a Boolean depending on whether or not the item is in the list.

Finding the index of an item

```
>>> symbols = [ 'H', 'He', 'Li', 'Be' ]
```

Finding the index is a method

the item to find

```
>>> symbols.index('H')
0
```

returns index of item

```
>>> metals = [ 'silver', 'gold', 'mercury', 'gold' ]
>>> metals.index('gold')
1
```

returns index of *first* matching item

66

The `index()` method returns the **index** of the *first* matching item in a list (there must be at least one matching item or you will get an error). For example:

```
>>> data = [2, 3, 5, 7, 5, 12]
>>> data.index(5)
2
```

There are also some other methods of lists which may be of interest. The `remove()` method **remove**s the *first* matching item from a list (there must be at least one matching item or you will get an error). As with `append()`, `remove()` works by modifying the list itself, not by creating a new list with one fewer item. For example:

```
>>> symbols = ['H','He','Li','Be','B','Li']
>>> symbols.remove('Li')
>>> symbols
['H', 'He', 'Be', 'B', 'Li']
```

The `insert()` method **insert**s an item into a list at the given position. It takes two arguments: the first is the index of the list at which the item is to be inserted, and the second argument is the item itself. As with `append()`, `insert()` works by modifying the list itself, not by creating a new list with an extra item. For example:

```
>>> symbols = ['H','He','Be','B']
>>> symbols.insert(2,'Li')
>>> symbols
['H', 'He', 'Li', 'Be', 'B']
```

Lists have a number of other methods, and appended to these notes is a guide to many of them.

Functions that give lists: **range()**

>>> **range(0, 8)**

[0, 1, 2, 3, 4, 5, 6, 7]

First integer in list

One beyond last integer in list
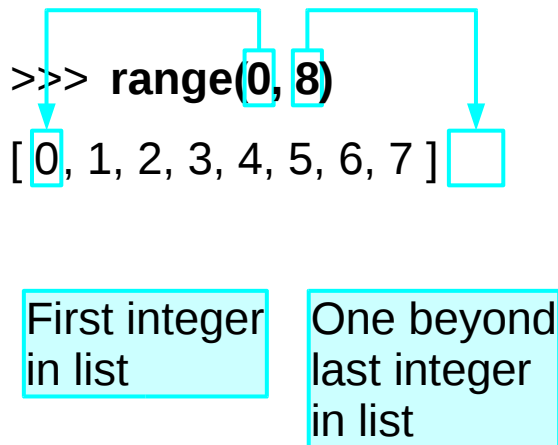
We have been happily manipulating lists. However, apart from entering them all manually, how do we get them? We cover this next as we look at some of the functions that return lists as their results.

One of the simplest built-in functions to give a list is the function `range()`. This takes two integers and gives a list of integers running from the first argument to one below the second.

range(): Why miss the last number?

same argument

>>> **range(0, 8)** + **range(8, 12)**

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

range(0, 12)
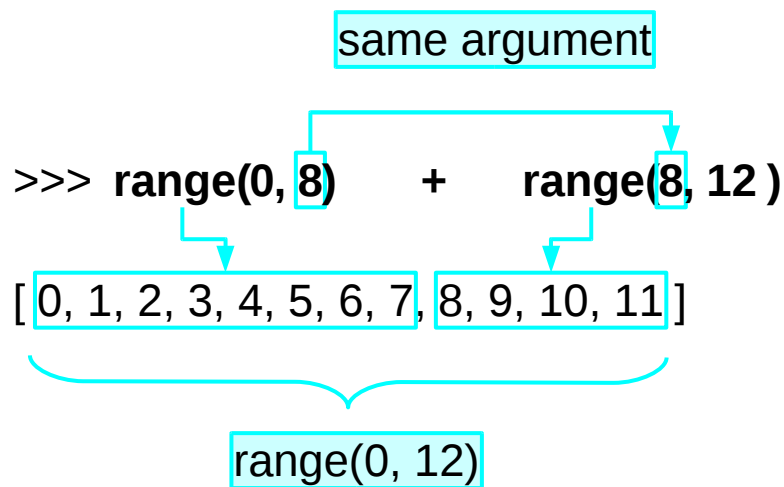
This business of "stopping one short" seems weird at first glance, but does have a purpose. If we concatenate two ranges where the first argument of the second range is the second argument of the first one then they join with no duplication of gaps to give a valid range again.

Functions that give lists: **split()**

original string

method built in to strings

```
>>> 'the cat sat on the mat'.split()
[ 'the', 'cat', 'sat', 'on', 'the', 'mat' ]
```

Split on white space

Spaces discarded

69

Another very useful function that gives a list is "splitting". This involves taking a string and splitting it into a list of sub-strings. The typical example involves splitting a phrase into its constituent words and discarding the spaces between them.

Note that `split()` is a method of strings. It returns a list of strings which are the words in the original string, with the white space between them thrown away. It is possible to use `split()` to chop on other than white space. You can put a string in the arguments of `split()` to tell it what to cut on (rather than white space) but we advise against it. It is very tempting to try to use this to split on commas, for example. There are much better ways to do this and we address them in the course "Python: Regular Expressions". Stick to splitting on white space.

For details of the "Python: Regular Expressions" course, see:

`http://training.csx.cam.ac.uk/course/pythonregexp`

`split()`: Only good for trivial splitting

>>> **'the cat sat on the mat'.split()**

['the', 'cat', 'sat', 'on', 'the', 'mat']

Split on white space

Spaces discarded

Trivial operation

Regular expressions

Comma separated values

Use the specialist
Python support
for these.

70

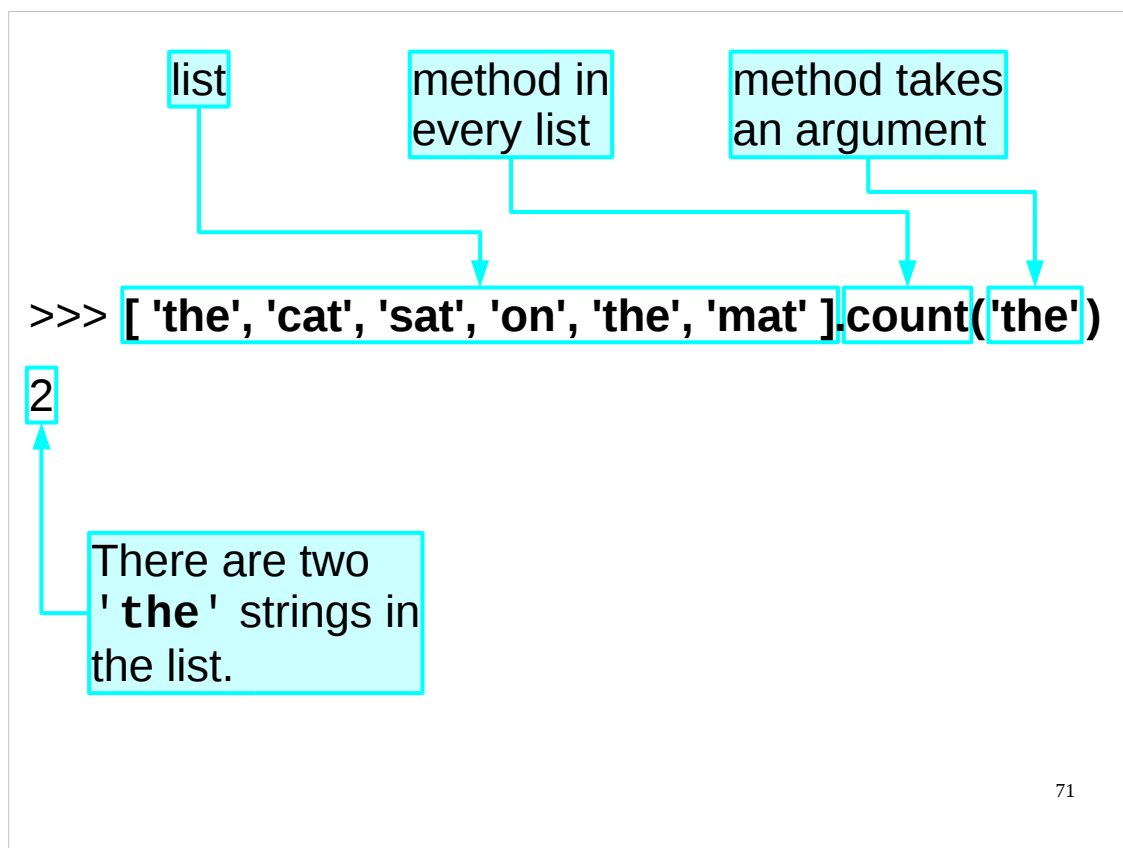To re-iterate: the `split()` method is very, very primitive.

There are many better approaches, such as using Python's support for regular expressions or for CSV (*c*omma *s*eparated *v*alues) files.

Python's support for CSV files is covered in the "Python: Further Topics" course. For details of this course, see:

> `http://training.csx.cam.ac.uk/course/pythonfurther`

For details of the "Python: Regular Expressions" course, which covers the use of regular expressions in Python, see:

> `http://training.csx.cam.ac.uk/course/pythonregexp`

```
                list         method in       method takes
                             every list      an argument

>>> [ 'the', 'cat', 'sat', 'on', 'the', 'mat' ].count('the')

2
```

There are two
`'the'` strings in
the list.

We'll divert very briefly from looking at functions that give lists to explore the "methods" idea a bit further. As we've seen, Python lists are *bona fide* objects with their own set of methods. One of these, for example, is "count()" which takes an argument and reports back how many times that item occurs in the list.

Combining methods

```
>>> 'the cat sat on the mat'.split().count('the')
2
```

First run **split()** to get a list

Second run count(**'the'**) on that list

We can run the split() and the count() methods together. If we are interested in seeing how often the word "the" occurs in a string we can split it into its constituent words with split() and then count the number of times the word "the" occurs with count(). But we don't need to create a variable to refer to the list; we can just run together the two methods as shown above.

(Note that strings also have a "count()" method which does something completely different. The count() method for strings takes an argument and reports how many times that argument occurs as a sequence of characters (a "sub-string") in the string. For example:
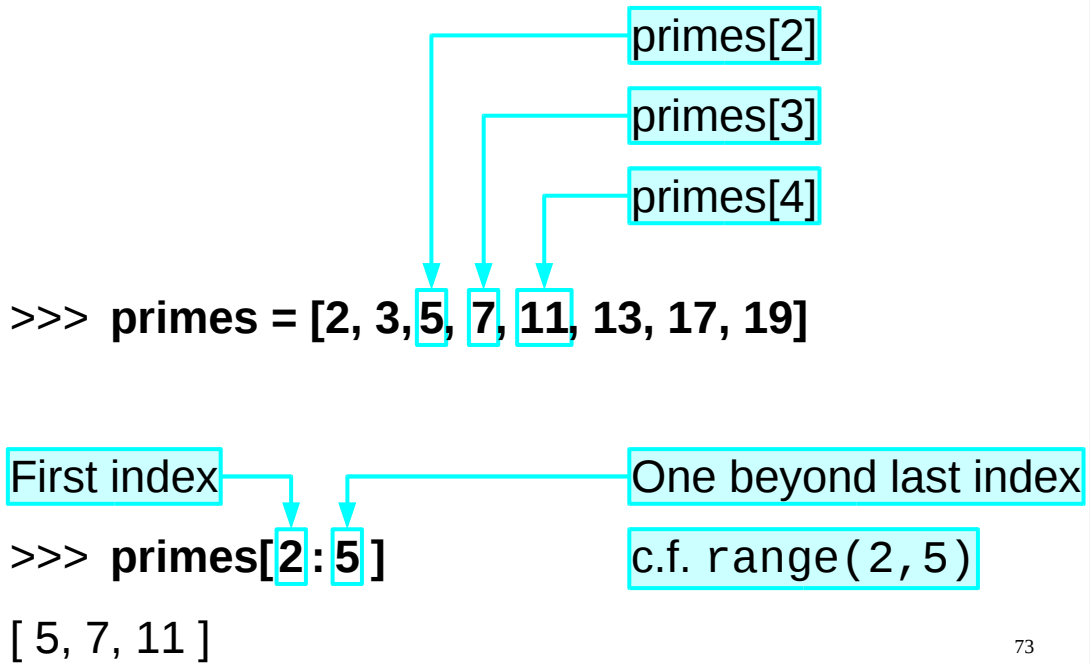```
>>> 'the cat sat on the mat'.count('at')
3
```
Because we want to find the number of times the *word* "the" occurs, we need to first split the string into a list of words and then use the count() method of that list. If we just used the count() method of the string then we would get the number of times the sub-string "the" occurred in the string, which might be more than the number of times the word "the" occurs, as below:
```
>>> 'three of the cats are over there'.count('the')
2
>>> 'three of the cats are over there'.split().count('the')
1
```
)

# Extracts from lists: "slices"

primes[2]

primes[3]

primes[4]

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

First index

One beyond last index

```
>>> primes[2:5]
```

c.f. `range(2,5)`

[5, 7, 11]

73

A very common requirement is to extract a sub-list from a list. Python calls these "slices" and they are the last of the things returning lists that we will consider.

The syntax for extracting a sub-list is very similar to that for extracting a single item. Instead of using a single index, though, we use a pair separated by a colon. The first index is the index of the first item in the slice. But the second index is the one *beyond* the last item in the slice. In this way it is very similar to the `range()` function we saw earlier, and, just as with `range()`, this quirk gives Python the property that slices can be concatenated in an elegant manner.

>>> **primes[2:5]**   Both limits given
[5, 7, 11]

>>> **primes[:5]**   Upper limit only
[2, 3, 5, 7, 11]

>>> **primes[2:]**   Lower limit only
[5, 7, 11, 13, 17, 19]

>>> **primes[:]**   Neither limit given
[2, 3, 5, 7, 11, 13, 17, 19]

74

Slices don't actually need both indices defined. If either is missing, Python interprets the slices as starting or ending at the start or end of the original list. If both are missing then we get a copy of the whole list. (In fact, the way to copy a list in Python is to take a slice of the list with both indices missing.)

```
#!/usr/bin/python

# This is a list of some metallic
#  elements.
metals = [ 'silver', 'gold', … ]

# Make a new list that is almost
#  identical to the metals list: the new
#  contains the same items, in the same
#  order, except that it does *NOT*
#  contain the item 'copper'.




# Print the new list.

                                   metals.py
```

What goes here?

Time for some more exercises.

In your course home directories you will find an incomplete script called `metals.py`.

Complete the script so that it takes the list `metals` and produces a new list that is identical to `metals` except that the new list should not contain any `'copper'` items. Then print out the new list.

Obviously, one way of doing this would be to manually type the items from the `metals` list, except for `'copper'`, into a new list. That would not be a very sensible way of doing things, and you wouldn't learn very much by doing it that way. I suggest you try something else.

If you have any questions about anything we've covered so far, now would be a good time to ask them.

```
#!/usr/bin/python

# This is a list of some data values.
data = [ 5.75, 8.25, … ]

# Make two new lists from this list.
#  The first new list should contain
#  the first half of data, in the same
#  order, whilst the second list should
#  contain the second half, so:
#     data = first_half + second_half
#  If there are an odd number of items,
#  make the first new list the larger
#  list.



# Print the new lists.
```

What goes here?

`data.py`

When you've finished the previous exercise, here's another one for you to try.

In your course home directories you will find an incomplete script called `data.py`.

Complete the script so that it takes the list `data` and produces two new lists. The first list should contain the first half of the `data` list, while the second list should contain the second half of the `data` list. (If `data` contains an odd number of items, then the first new list should be the larger list.) Then print out the new lists.

Obviously, one way of doing this would be to manually type the items from `data` into two new lists. That would not be a very sensible way of doing things, and you wouldn't learn very much by doing it that way.

If you have any questions about either this exercise or the previous one, or about anything we've covered so far, now would be a good time to ask them.

```python
#!/usr/bin/python

# This is a list of some metallic
#  elements.
metals = [ 'silver', 'gold', … ]

# Make a new list that is almost
#  identical to the metals list: the new
#  contains the same items, in the same
#  order, except that it does *NOT*
#  contain the item 'copper'.
new_metals = []
for metal in metals:
    if metal != 'copper':
        new_metals.append(metal)

# Print the new list.
print new_metals
```

metals.py

77

Here is one solution to the first exercise.

It's not the only possible solution, but it is quite a nice one because it works regardless of the number of times `'copper'` appears in the `metals` list.

If, as in the example `metals` list given, the item you want to remove only appears once, you could use either of the solutions below to create the `new_metals` list.

One possible solution is to make two list slices, one on either side of the item we wish to remove (`'copper'`):

```python
bad_index = metals.index('copper')
new_metals = metals[:bad_index] + metals[bad_index+1:]
```

Can you see why the above solution works even if `'copper'` is the first or last item in the `metals` list?

Another possible solution is to make a copy of the list and then remove the unwanted item using the `remove()` method of lists:

```python
new_metals = metals[:]
new_metals.remove('copper')
```

(Note that we need to make an *independent* copy (a so-called "deep copy") of the `metals` list, so we need to make a slice of the entire `metals` list ("`metals[:]`") rather than using "`new_metals = metals`".)

If there is anything in any of the above solutions that you don't understand, or if your answer was wildly different to all of the above solutions, please let the course giver know now.

```python
#!/usr/bin/python

# This is a list of some data values.
data = [ 5.75, 8.25, … ]

# Make two new lists from this list.
#  The first new list should contain
#  the first half of data, in the same
#  order, whilst the second list should
#  contain the second half, so:
#    data = first_half + second_half
#  If there are an odd number of items,
#  make the first new list the larger
#  list.
if len(data) % 2 == 0:
    index = len(data) / 2
else:
    index = (len(data) + 1) / 2

first_half = data[:index]
second_half = data[index:]

# Print the new lists.
print first_half
print second_half
```

Answer

data.py

78

And above is a solution to the second exercise I asked you to try. It's not the only possible solution, but I've chosen this one because it is one of the most straightforward solutions.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

Dictionaries

Key ➡ Value

dictionary
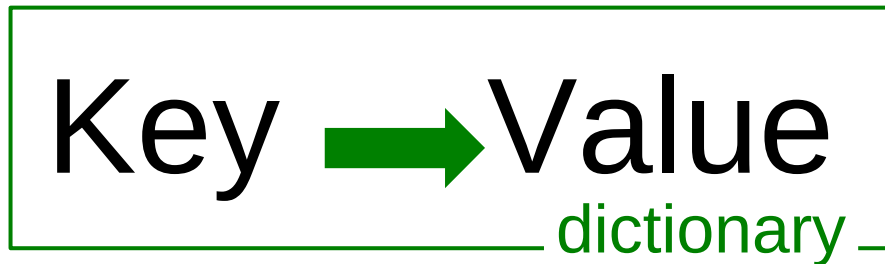
79

For every list – a sequence of items – there is a way we can say "item zero", "item one" and so on. The fact that they are in a sequence implies an index number which in turn can be used to identify individual items in the list. We could regard lists as a mapping from the numbers 0, 1, … to the items in the list:

$$\{0, 1, \dots\} \rightarrow \{\text{items in list}\}$$

We can give the list an integer and it will returns the item whose index corresponds to that integer.

We can generalise this idea. Consider a list not as a sequence that implies an index, but rather as a disorganised set of items whose individual items can be identified by a number. So the "index" is no longer implied by any internal ordering but is an explicit part of the construct. We have some object that takes a number and hands back an item. In this case what was called an "index" is now called a "key". Keys are explicit parts of the object, not numbers implied by the internal structure of the object.

Once we have made this jump then there is no reason for the keys to be numbers at all. Instead of "give me the item corresponding to the integer 2" we can ask "give me the item corresponding to the string 'He'" (for example).

This general mapping from some type to another type is called a "dictionary" in Python. A dictionary maps a "key" (which can be almost any simple data type) to a "value" (which can be any data type):

$$\{\text{keys}\} \rightarrow \{\text{values}\}$$

These are the objects we will be considering next.

The jargon term for this sort of structure is an "associative array", and many modern programming languages have an "associative array" type. As just mentioned, in Python, these are known as "dictionaries". In Perl they are called "hashes". In Java and C++ these are known as "maps". In Visual Basic there is no standard implementation of this type that is common to all versions of Visual Basic (in Visual Basic 6.0 you can use the Scripting.Dictionary object, whilst in Visual Basic.NET you use the collection classes from the .NET Framework).
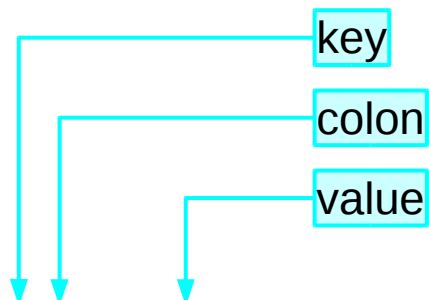
Creating a dictionary — 1

curly brackets

dictionary entry

dictionary entry

comma

```
>>> names =  {'H': 'hydrogen', 'He': 'helium'}
```
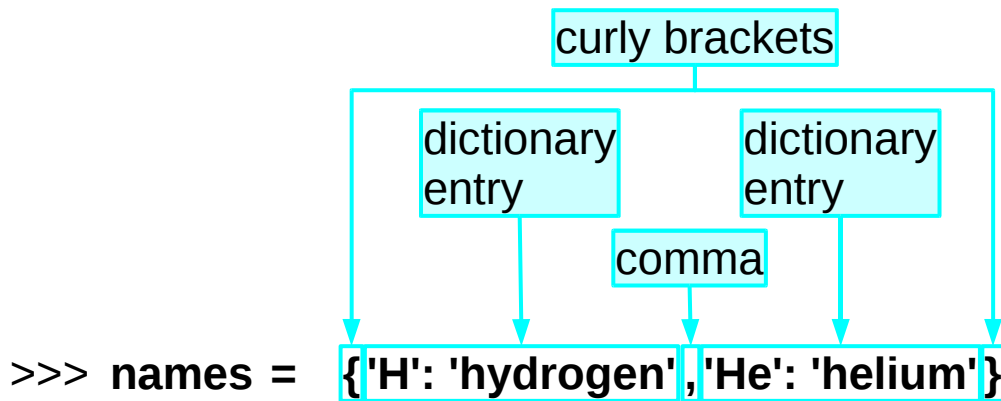
We can create a dictionary all in one go by listing all the key→value pairs in a manner similar to a list but with curly brackets ("braces") around them.

The convention we're using here is to name the dictionary after the *values* it contains (not the keys). This is just a convention and nothing like as common as the convention for lists. The authors' preferred convention is to call the dictionary `symbol_to_name` or something like that, but long names like that don't fit on slides.
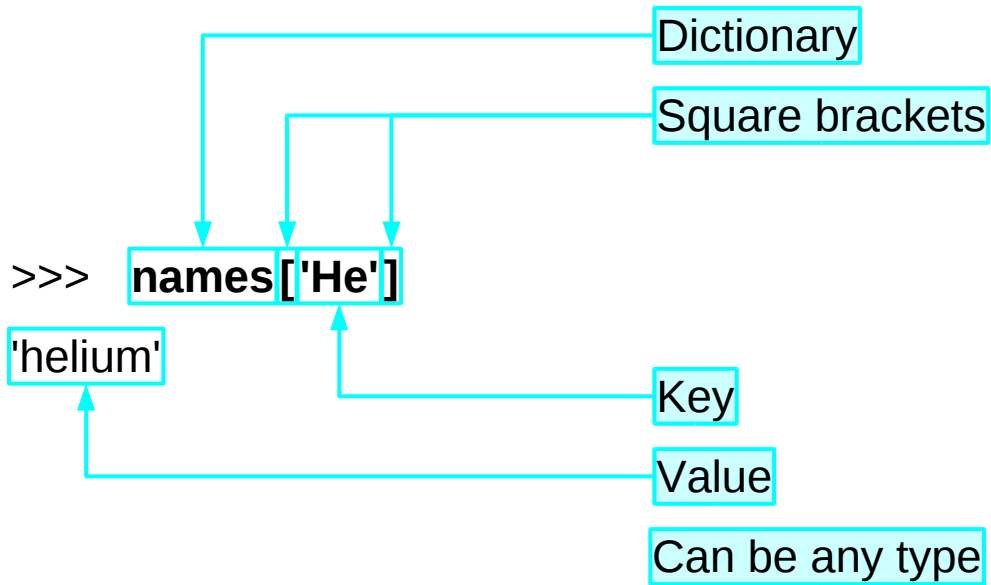
Creating a dictionary — 2

key

colon

value

>>> **names = { 'H' : 'hydrogen' , 'He': 'helium'}**

The individual key → value pairs are separated by a colon.

Accessing a dictionary

Dictionary

Square brackets

>>> **names['He']**

'helium'

Key

Value

Can be any type

82

To get at the value corresponding to a particular key we do have the same syntax as for lists. The key follows the dictionary surrounded by square brackets.

Given this repeated similarity to lists we might wonder why we didn't use square brackets for dictionary definition in the first place. The answer is that the interpreter would then not be able to distinguish the empty list ("[ ]") from the empty dictionary ("{}").

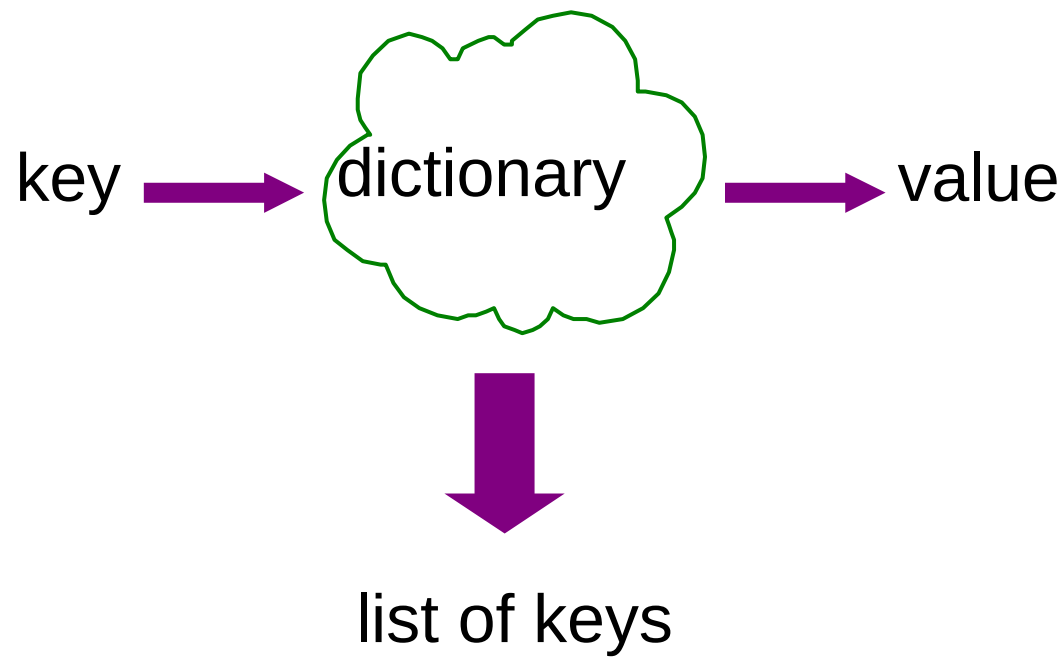# Creating a dictionary — 3

>>> **names = {}** ← Start with an empty dictionary

>>> **names['H'] = 'hydrogen'**

>>> **names['He'] = 'helium'**   Add
                                        entries
>>> **names['Li'] = 'lithium'**

>>> **names['Be'] = 'beryllium'**

83

The other way to create a dictionary is to create an empty one and then add key→value items to it one at a time.

So we have a dictionary which turns keys into values. Therefore it must know all its keys. So how do we get at them?

Dictionaries can be easily persuaded to hand over the list of their keys.

Treat a dictionary like a list…

Python expects a list here

for symbol in names :

print symbol , names[symbol]

del symbol

Dictionary key

…and it behaves like a list of keys

85

The basic premise is that if you plug a dictionary into any Python construction that requires a list then it behaves like a list of its keys. So we can use dictionaries in `for` loops like this.

Note that we use the variable name "`symbol`" simply because it makes sense as a variable name; it is not a syntactic keyword in the same way as "`for`" and "`in`" (or even "`:`"). The script fragment

```
for symbol in names:
    print symbol, names[symbol]
```

is exactly the same as

```
for long_variable_name in names:
    print long_variable_name, names[long_variable_name]
```

except that it's shorter and easier to read!

Note that we name the variable `symbol`, which will be carrying the values of keys in the dictionary, after the keys, not the values. As the `names` dictionary has the symbols of atomic elements as *keys* and the names of atomic elements as values, we name the variable `symbol` after the *symbols* of atomic elements.

If we just want to get a list of all the keys of a dictionary, we can use the `keys()` method of the dictionary, which returns a list of the keys in the dictionary (in no particular order).

```
>>> names = {'H':'hydrogen','He':'helium'}
>>> names.keys()
['H', 'He']
```

Example

```
#!/usr/bin/python

names = {
   'H': 'hydrogen',
   'He': 'helium',
   …
   'U':  'uranium',
   }

for symbol in names:
   print names[symbol]
del symbol
```
chemicals.py

$ **python chemicals.py**

ruthenium
rhenium
…
astatine
indium

No relation between order in file and output!

Note that there is no implied ordering in the keys. If we treat a dictionary as a list and get the list of keys then we get the keys in what looks like a random order. It is certainly not the order they were added in to the dictionary.

Missing keys

missing key

>>> **names['Np']**

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Np'

Type of error

Missing key

What happens if you ask for a key that the dictionary does not have? Just as there is an error for when a list is given an out of range index, the dictionary triggers an error if it is given an unknown key.

Treat a dictionary like a list…

Python expects a list here

if symbol in names :

print symbol , names[symbol]

…and it behaves like a list of keys

Ideally, we would be able to test for whether a key is in a dictionary. We can.

The Python "in" keyword can be used to test whether an item is in a list or not. This involves treating something like a list so we can get the same effect with a dictionary. The "in" keyword will test to see if an item is a *key* of the dictionary.

There is no corresponding test to see whether an item is a valid *value* in a dictionary.

However, there is a sneaky way we can do this. Dictionaries have a values() method, which returns a list of the values of the dictionary (in no particular order). We can then use the "in" keyword to see whether an item is in this list. If so, then it is a valid *value* in the dictionary.

```
>>> names = {'H':'hydrogen','He':'helium'}
>>> names.values()
['hydrogen', 'helium']
>>> 'helium' in names.values()
True
```

(At this point, those of you who are overly impressed with your own cleverness may think you that you can use the keys() and values() methods together to get two lists where the order of the items in the list from keys() is related to the order of the items in the list from values(), i.e. the first item in the list from keys() is the key in the dictionary whose value is the first item in the list from values(), and so on. Be aware that this is *not* always true. **So don't rely on it.**)

Missing keys

>>> **names['Np']**

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Np'

>>> **'Np' in names** ← Test for membership of a *list*

False ← 'Np' is not a key in the dictionary

So, as for lists, we get a simple Boolean (`True` or `False`) when we test for key membership.

```
>>> names = {'H': 'hydrogen', 'He': 'helium', 'Li':
'lithium'}
>>> 'Np' in names
False
>>> 'Li' in names
True
```

Note that the "`names =`" line above is a single line and should be typed in as such – it is split across two lines here to make it easier to read.

And now for something completely…

Obviously when you create a dictionary you need to be clear about which items are the keys and which are the values. But what if you are given a dictionary that is the "wrong way round"?

Have a look at the script `chemicals_reversed.py` in your home directory.

See if you can figure out what it does – if there is anything you don't understand, tell the course giver or demonstrator. 90

To see dictionaries in practice, stop for a while and inspect a Python script we have written for you. This script takes a dictionary that resolves symbols of atomic elements to names of atomic elements and creates a dictionary that takes names of atomic elements and gives their symbols. Examine the script and see if you can figure out what it does (and why it does it). The script is called `chemicals_reversed.py` and is in your course home directories.

If you have any questions, please ask the course giver or a demonstrator. After you've finished looking at this script, take a break.

In case there was anything in the script you didn't understand, this is what it does:

Start: We start with the "right way round" dictionary and we name this dictionary, which maps symbols of atomic elements to their names, `names`.

```
names = {…}
```

Create empty dictionary: Next we create the empty "reversed" dictionary which maps names of atomic elements to the corresponding symbols, `symbols`:

```
symbols = {}
```

Loop: Then we need to fill `symbols`. We know how to do a `for` loop, so we'll do one here to fill in the reversed dictionary, `symbols`, one item at a time. We name the loop variable after the keys it steps through (hence "`symbol`").

```
for symbol in names:
```

Look up name of atomic element: In the "indented block" that is run once for each key we will look up the key (symbol of the atomic element) in the `names` dictionary to get the name of the atomic element.

```
name = names[symbol]
```

Add reversed entry to dictionary: We will then assign it in reversed fashion into `symbols`.

```
symbols[name] = symbol
```

Clean up: Once out of the loop (and therefore unindented) we will `del` the variables we used for the loop because we're good programmers who write clean, hygienic code.

```
del name
del symbol
```

Print the dictionary: Finally, now that we have the "reversed" dictionary, we print it out.

Defining functions

define a function ──────▶ def reverse (a_to_b) :

Values in: a_to_b

Values out: b_to_a

Internal values: a b

Internal values
are automatically
cleaned up on exit.

```
def reverse ( a_to_b ) :
    b_to_a = {}
    for a in a_to_b :
        b = a_to_b[a]
        b_to_a[b] = a
    return b_to_a
```

As with most programming languages, Python allows us to define our own functions.

So let's examine what we want from a function and how it has to behave.

Our function will have a well-defined, and typically small, set of inputs which will be passed to it as its arguments.

Similarly, we want a well-defined set of outputs. Python functions are permitted a single output only, though as we will see shortly, it is standard practice to pass several values bundled together in that single output.

So we have well-defined inputs and outputs. What about variables?

Python works like this: If any variables are created or updated inside the function then they are automatically made local to that function. Any external variables of the same name are ignored. If a variable is read, but not changed then Python will first look for a local variable and, if it doesn't exist, then go looking for an external one.

How do we define a function?

To define a function we start with the keyword "def" to announce the **def**inition of a function. This is followed by the name of the function being defined. In the example above the function is called "reverse".

Then comes the specification of the input[s] required by the function. These appear just as they do when the function is used: in round brackets. If there is more than one input to the function then they are separated by commas. The whole line ends with a colon…

…and after a colon comes indentation. The body of our function will be indented in the Python script.

(Note how every single variable is local to the function either because it was passed in as a function argument (a_to_b) or because it was created inside the function (b_to_a, a, b).)

Finally, we need to get the desired value out of the function. We do this with the "return" statement. This specifies what value the function should return and marks the end of the function definition.

Example

```
#!/usr/bin/python

def reverse(a_to_b):
  b_to_a = {}
  for a in a_to_b:
    b = a_to_b[a]
    b_to_a[b] = a
  return b_to_a

names = {…}

symbols = reverse(names)
…          chemicals2.py
```
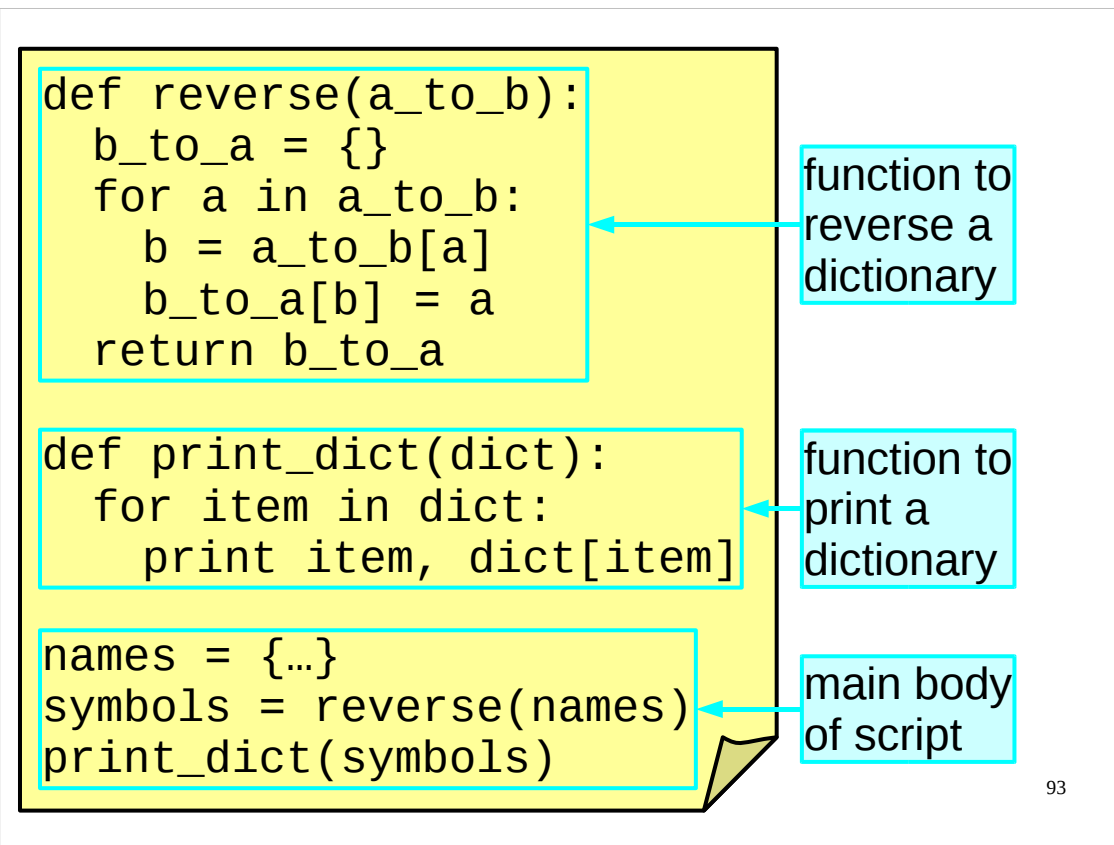
We've put this function in the script `chemicals2.py` in your course home directories. You can try it out and compare it to the `chemicals_reversed.py` script.

In this script we have taken the main body of the `chemicals_reversed.py` script and turned it into functions.

```
def reverse(a_to_b):
    b_to_a = {}
    for a in a_to_b:
        b = a_to_b[a]
        b_to_a[b] = a
    return b_to_a
```

function to reverse a dictionary

```
def print_dict(dict):
    for item in dict:
        print item, dict[item]
```

function to print a dictionary

```
names = {…}
symbols = reverse(names)
print_dict(symbols)
```

main body of script

Note that the function definitions are at the start of the script. It is traditional that function definitions appear at the start of a script; the only requirement is that they are defined before they are used.

Where we used to have the functionality that created `symbols` by reversing `names` we now have a single line that calls the function

```
symbols = reverse(names)
```

The printing functionality has been replaced by another single line that calls the function to print the new dictionary:

```
print_dict(symbols)
```

Please note that the line

```
symbols = reverse(names)
```

means "pass the value currently held in the external variable `names` into the function `reverse()` and take the value returned by that function and stick it in the variable `symbols`".

Let's try it out…

```
#!/usr/bin/python

def reverse(a_to_b):
   b_to_a = {}
   for a in a_to_b:
      b = a_to_b[a]
      b_to_a[b] = a
   return b_to_a

names = {…}

symbols = reverse(names)
…

        chemicals2.py
```

$ **python chemicals2.py**
gold Au
neon Ne
cobalt Co
germanium Ge
…
tellurium Te
xenon Xe

94

Run the chemicals2.py script and see what it does. You should find it behaves exactly the same as the chemicals_reversed.py script.

Re-using functions: "modules"

Two functions:

```
reverse(a_to_b)
print_dict(dict)
```
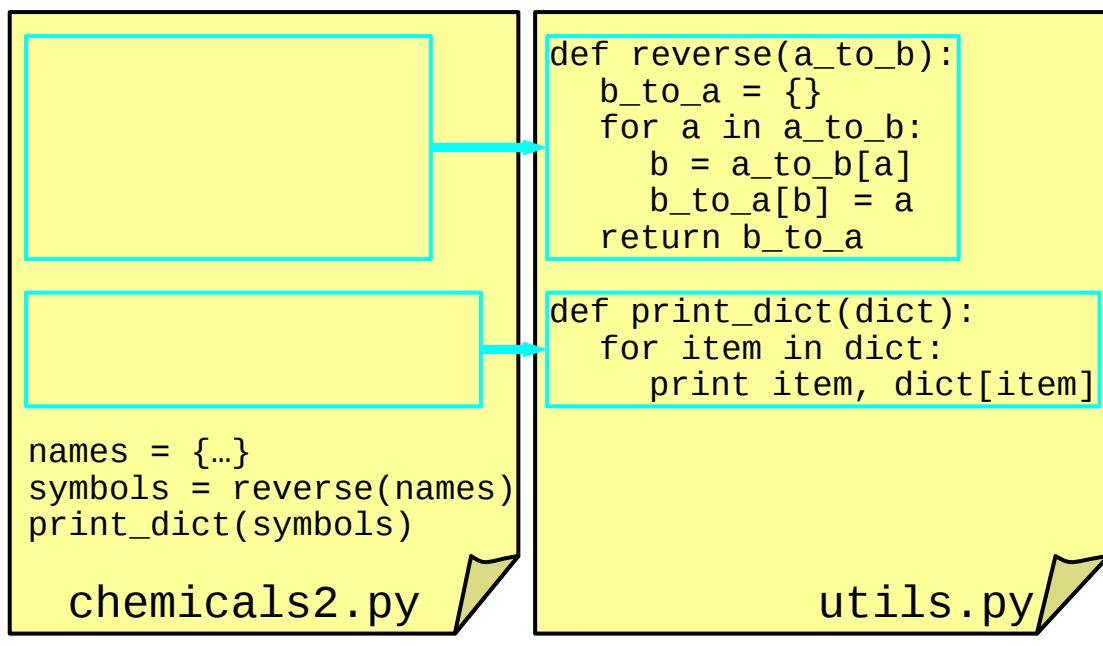
currently in `chemicals2.py`

We have a function that can reverse any one-to-one dictionary (i.e. a dictionary where no two keys give the same corresponding value) and another function that prints out the key → value pairs of any dictionary. These are general purpose functions, so wouldn't it be nice to be able to use them generally? We can reuse a function within a script, but what about between scripts?

In Python, we re-use functions by putting them in a file called a "*module*". We can then load the module and use the functions from any script we like. Modules are Python's equivalent of libraries in other programming languages.

Modules — 1

Put function definitions to new file `utils.py`

```
                                    def reverse(a_to_b):
                                        b_to_a = {}
                                        for a in a_to_b:
                                            b = a_to_b[a]
                                            b_to_a[b] = a
                                        return b_to_a

                                    def print_dict(dict):
                                        for item in dict:
                                            print item, dict[item]


names = {…}
symbols = reverse(names)
print_dict(symbols)

   chemicals2.py                              utils.py
```

So let's do precisely that: use our function in different scripts.

We start by
(a)   opening a new file called `utils.py`,
(b)   cutting the definitions of the two functions from `chemicals2.py`,
(c)   pasting them into `utils.py`, and
(d)   saving both files back to disc.

The script `chemicals2.py` no longer works as it uses functions it doesn't know the definitions for:

```
$ python chemicals2.py
Traceback (most recent call last):
  File "chemicals2.py", line 98, in <module>
    symbols = reverse(names)
NameError: name 'reverse' is not defined
```
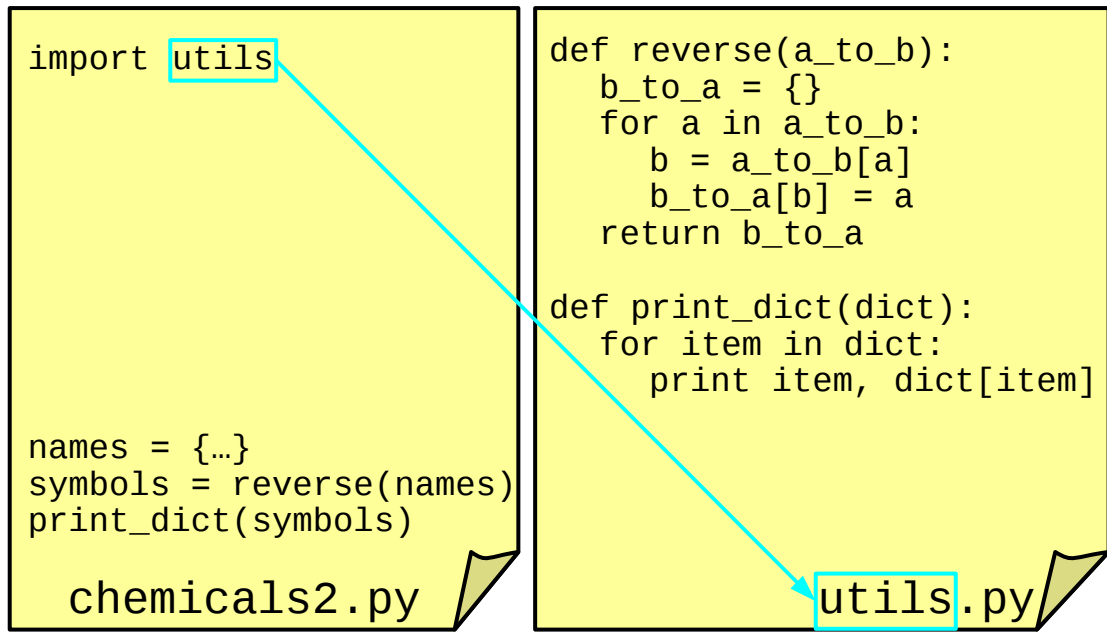
So we need to connect `chemicals2.py` with `utils.py`.

Modules — 2

"import" the module

```
import utils




names = {…}
symbols = reverse(names)
print_dict(symbols)

    chemicals2.py
```

```
def reverse(a_to_b):
    b_to_a = {}
    for a in a_to_b:
        b = a_to_b[a]
        b_to_a[b] = a
    return b_to_a

def print_dict(dict):
    for item in dict:
        print item, dict[item]




            utils.py
```

We replace the definitions with one line, "`import utils`", which is the instruction to read in the definitions of functions in a file called `utils.py`.

Our script still doesn't work.

If you try you should get the same error message as before (with a different line number) but if you have mistyped the `import` line you will get this error for mistyping "`import`":

```
$ python chemicals2.py
  File "chemicals2.py", line 3
    imoprt utils
                ^
SyntaxError: invalid syntax
```

and this error for mistyping "`utils`":
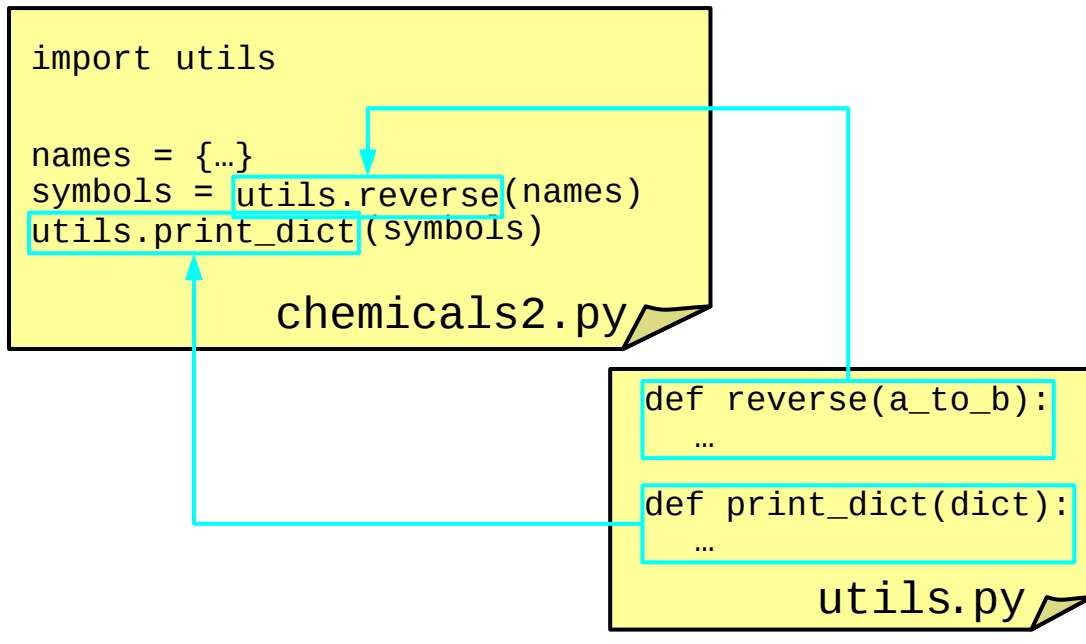
```
$ python chemicals2.py
Traceback (most recent call last):
  File "chemicals2.py", line 3, in <module>
    import uitls
ImportError: No module named uitls
```

Note the use of the word "module". We will see that again soon.

You may wonder how Python knows where to find the `utils.py` file. We'll return to this later. For now you just need to know that, unless someone has configured Python to behave differently, Python will search for any file you ask it to `import` in the current directory (or the directory containing the script you are running when you are running a Python script), and, if it can't find the file there, it will then try some system directories.

Modules — 3

Use functions from the module

```
import utils


names = {…}
symbols = utils.reverse(names)
utils.print_dict(symbols)
```

chemicals2.py

```
def reverse(a_to_b):
    …
```

```
def print_dict(dict):
    …
```

utils.py

The problem is that Python is still looking for the function definitions in the same file (`chemicals2.py`) as it is calling them from. We need to tell it to use the functions from the `utils.py` file. We do that by prefixing "`utils.`" in front of every function call that needs to be diverted.

Now the script works:

```
$ python chemicals2.py
gold Au
neon Ne
cobalt Co
germanium Ge
…
manganese Mn
tellurium Te
xenon Xe
```

and we have a `utils.py` file that we can call from other scripts as well.

Let's check it still works...

```python
#!/usr/bin/python

import utils

names = {…}

symbols = utils.reverse(names)
utils.print_dict(symbols)
```
chemicals2.py

```
$ python chemicals2.py
gold Au
neon Ne
cobalt Co
germanium Ge
…
tellurium Te
xenon Xe
```

After you've moved the functions into the `utils.py` file and adapted the `chemicals2.py` script appropriately, you should run your modified `chemicals2.py` script and check that it still works.

The collection of functions provided by the file `utils.py` is known as the "utils *module*" and the operation performed by the line "`import utils`" is referred to as "importing a module".

When you ask Python to import a module, it first checks to see whether the module is one of the built-in modules that are part of Python. If not, it then searches in a list of directories and loads the **first** matching file it finds. Unless someone has configured Python differently, this list of directories consists of the current directory (or the directory in which your script lives when you are running a script) and some system directories. This list of directories is kept in the `sys` module in a variable called `path`. You can see this list by importing the `sys` module and then examining `sys.path`:

```
>>> import sys
>>> sys.path
['', '/usr/lib/python26.zip', '/usr/lib/python2.6', '/usr/lib/python2.6/plat-
linux2', '/usr/lib/python2.6/lib-tk', '/usr/lib/python2.6/lib-old',
'/usr/lib/python2.6/lib-dynload', '/usr/lib/python2.6/site-packages',
'/usr/lib/python2.6/site-packages/Numeric', '/usr/lib/python2.6/site-
packages/PIL', '/usr/local/lib/python2.6/site-packages',
'/usr/lib/python2.6/site-packages/gtk-2.0', '/usr/lib/python2.6/site-
packages/wx-2.8-gtk2-unicode']
```

(Note that in this context, the empty string, `''`, means "look in the current directory".) As `sys.path` is a Python list, you can manipulate it as you would any other Python list. This allows you to change the directories in which Python will look for modules (and/or the order in which those directories are searched).

You can also affect this list of directories by setting the `PYTHONPATH` environment variable before you start the Python interpreter or run your Python script. If the `PYTHONPATH` environment variable is set, Python will add the directories specified in this environment variable to the *start* of `sys.path`, *immediately after* the `''` item, i.e. it will search the current directory (or the directory containing the script when you are running a script), then the directories specified in `PYTHONPATH` and then the system directories it normally searches.
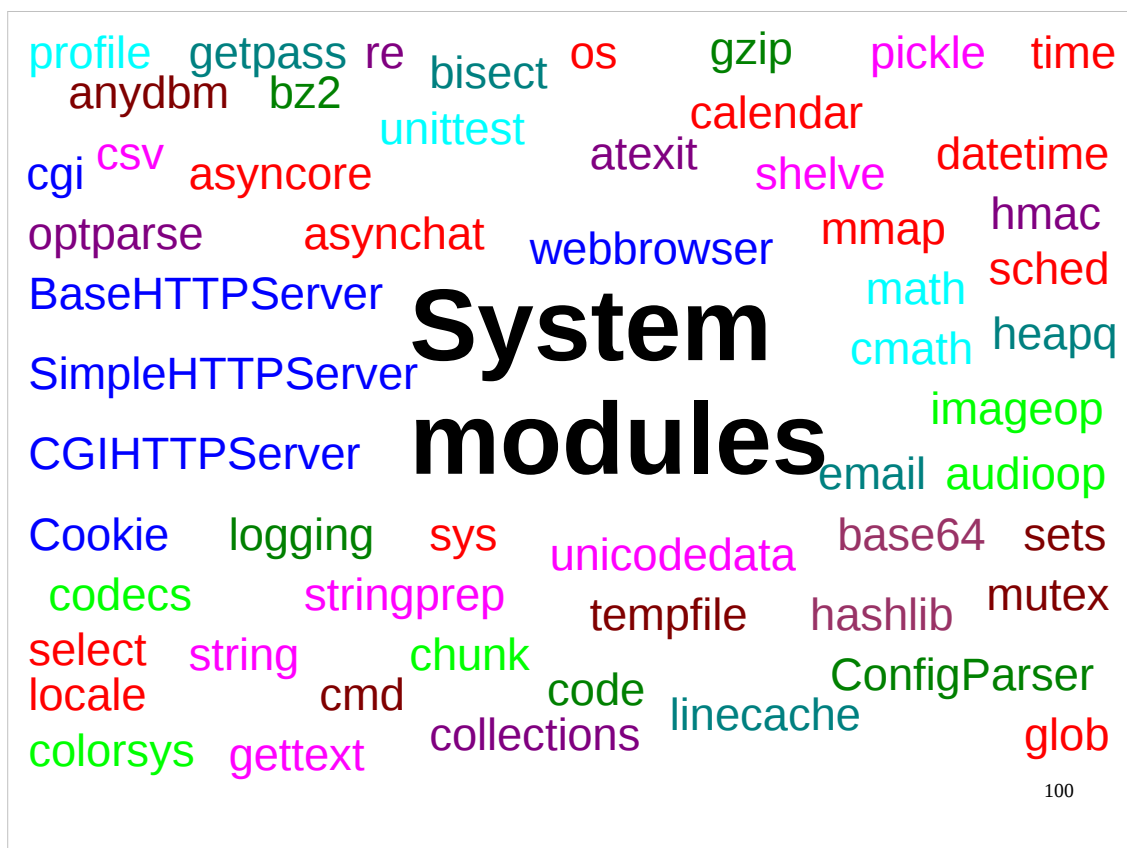
There are, of course, very many modules provided by the system to provide collections of functions for particular purposes.

Almost without exception, every course that explains how to do some particular thing in Python starts by importing the module that does it. Modules are where Python stores its big guns for tackling problems. Appended to the notes is a list of the most commonly useful Python modules and what they do.

On any given installation of Python you can find the names of all the modules that are available using the help() function:

```
>>> help('modules')

Please wait a moment while I gather a list of all available modules...

Alacarte               _LWPCookieJar          gconf                  pycompile
ArgImagePlugin         _MozillaCookieJar      gdbm                   pyclbr
…
XbmImagePlugin         functools              pty                    zope
XpmImagePlugin         gc                     pwd

Enter any module name to get more help. Or, type "modules spam" to search
for modules whose descriptions contain the word "spam".

>>>
```
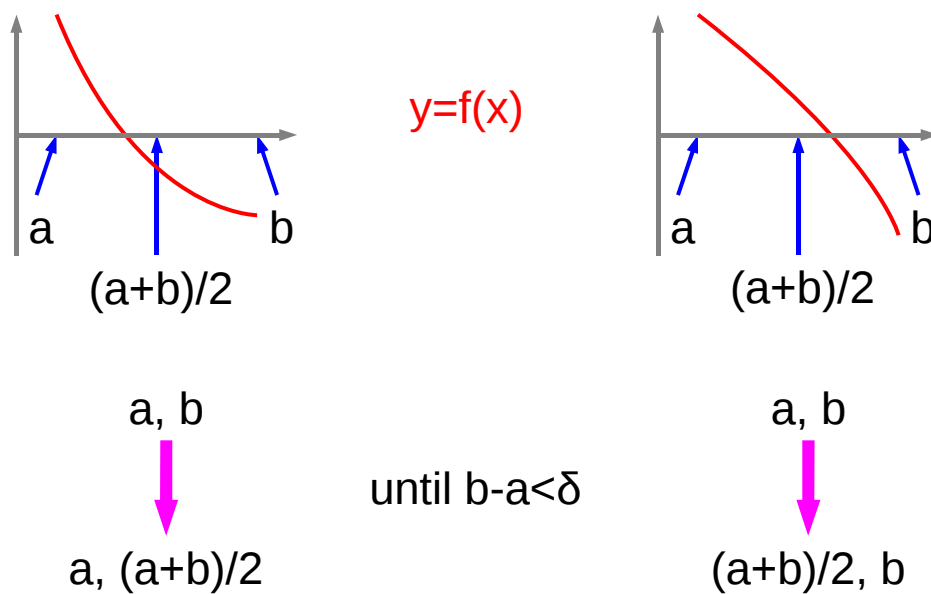
The complete list of the modules that ship with Python can be found in the Python documentation at the following URL:

http://docs.python.org/modindex.html

Root-finding by bisection

y=f(x)

a   b
(a+b)/2

a   b
(a+b)/2

a, b

until b-a<δ

a, b

a, (a+b)/2

(a+b)/2, b

Next we will look at getting more complex information into and out of our functions. Our functions to date have taken a single input argument and have returned either no output or a single output value. What happens if we want to read in or to return several values?
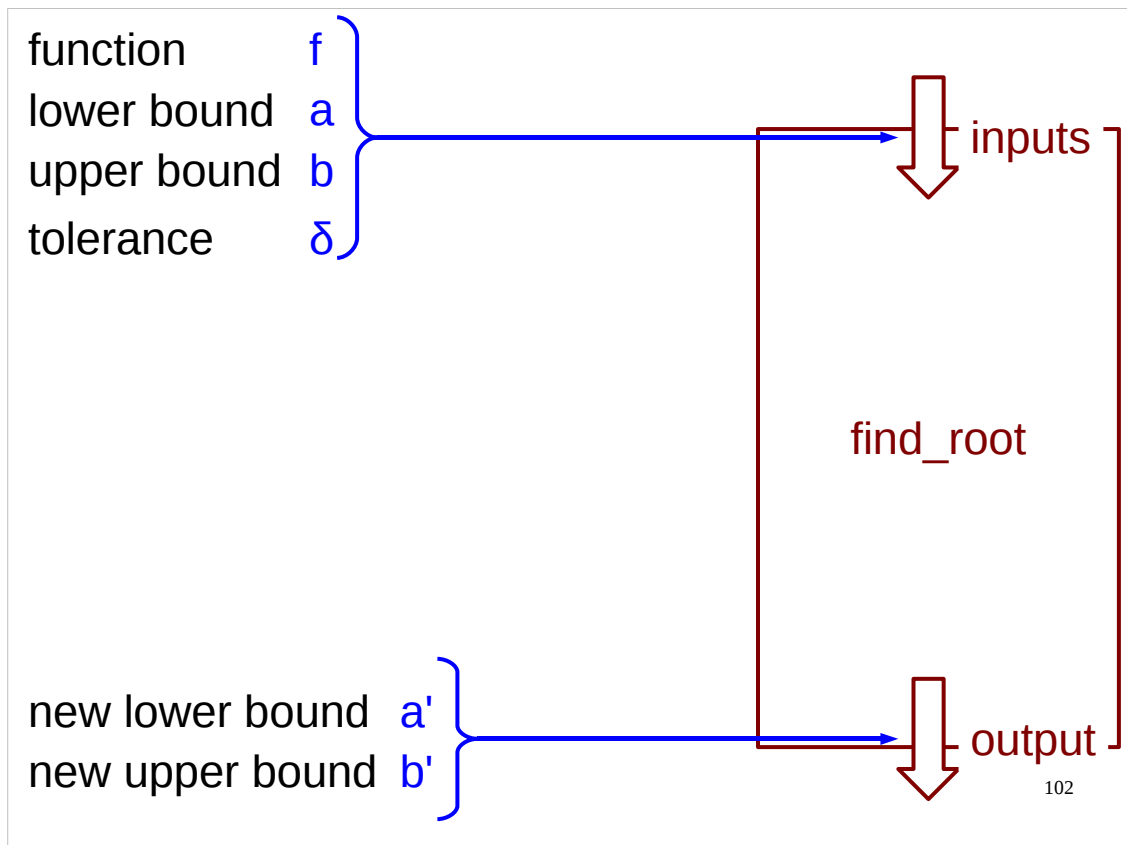
I'm picking a simple numerical technique as my example.

We want to find numerical approximations to the roots of (mathematical) functions. We know the root is in a particular interval if the function is positive at one end of the interval and negative at the other end. This means that the function is zero somewhere in between the ends of the interval. (This is the "intermediate value theorem" for the maths nargs reading.)

The trick is to cut the interval in two and calculate the function at the mid point. If it's positive then the root is between the midpoint and the end that has a negative value. If it's negative the root is between the midpoint and the end with the positive value.

There's a neat trick we can apply doing this. If the root lies between *(a+b)/2* and *b* then the function values at these two points have different signs; one is positive and one is negative. This means that their product is negative. If the root lies between *a* and *(a+b)/2* then the function is either positive at both *(a+b)/2* and *b* or negative at both these points. Either way, the product of the values at the two points is positive. So we can use the sign of *f(b)×f((a+b)/2)* as a test for which side of *(a+b)/2* the root is on.

We repeat this bisection until we have an interval that's close enough for us.

function   f

lower bound   a

upper bound   b

tolerance   δ

inputs

find_root

new lower bound a'

new upper bound b'

output
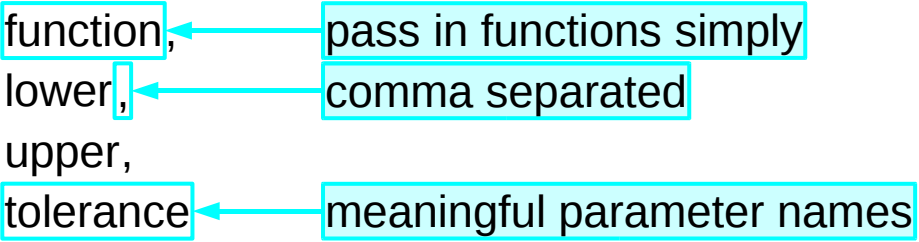
Our Python function will need four inputs, then:
- the mathematical function whose root we are looking for,
- the two ends of the interval, and
- some measure of how small the interval has to be to satisfy us.

And it will return two outputs:
- the two ends of the new interval.

Multiple values for input

```
def find_root(
    function,
    lower,
    upper,
    tolerance
    ):
```

function → pass in functions simply

lower → comma separated

tolerance → meaningful parameter names

function body

103

So how do we code this in Python? Changing to multiple inputs is easy; we simply list the inputs separated by commas. When we call the function we will pass in multiple values separated by commas in the same order as they appear in the definition.

Note that you can spread the arguments over multiple lines, using the standard Python indenting style. This lets you pick sensible variable names.

Note also that you can pass a function as input to another function just by giving its name. You don't have to use a special syntax to pass a function as input, nor do you need to mess around with pointers and other such strange beasts as you do in some other programming languages.

Multiple values for output

```
def find_root(
    …
):
```

    function body

```
return (
    lower ,
    upper
)
```

typically on a single line

So that was how to handle multiple inputs. What about multiple outputs?

The answer is simple. Exactly as we did for inputs we return a set of values in round brackets and separated by commas. As with the inputs, you can spread the arguments over multiple lines, using the standard Python indenting style.

So, instead of returning a single value we can return a pair.

```
def find_root(
    function,
    lower,
    upper,
    tolerance
    ):

    while upper - lower > tolerance:
        middle = (lower + upper) / 2.0
        if function(middle)*function(upper) > 0.0:
            upper = middle
        else:
            lower = middle

    return (lower, upper)
```

`utils.py`

So here is our completed function in our `utils.py` file. You should modify your copy of `utils.py` so that it has this function definition in it as it will be needed for the next example.

Don't get hung up on the details of the function, but notice that if it is using floating point numbers it should use them consistently and uniformly throughout.

Recall why we multiply the two function values together and check whether its positive or not:

If the root lies between *(a+b)/2* and *b* then the function values at these two points have different signs; one is positive and one is negative. This means that their product is negative. If the root lies between *a* and *(a+b)/2* then the function is either positive at both *(a+b)/2* and *b* or negative at both these points. Either way, the product of the values at the two points is positive. So we can use the sign of *f(b)×f((a+b)/2)* as a test for which side of *(a+b)/2* the root is on.

```
#!/usr/bin/python

import utils

def poly(x):
    return x**2 - 2.0

print utils.find_root(poly, 0.0, 2.0, 1.0e-5)
```

Find the root
of this function

sqrt2.py

$ **python sqrt2.py**
(1.4142074584960938, 1.414215087890625)

We wrote our root finder in `utils.py`, so we can quickly write a script to exploit it, and there is such a script in the file `sqrt2.py` in your course home directories. Note how we write a simple function in Python and pass it in as the first argument.

Recall that `x**2` means x squared (x^2).

Recall that our `find_root()` function returns a pair of values. So, what does a pair of values look like? Well, we can print it out and we get, printed, the two values inside parentheses and separated by a comma. Simple, really!

```
#!/usr/bin/python

import utils

def poly(x):
    return x**2 - 2.0

(lo, up) = utils.find_root(poly, 0.0, 2.0, 1.0e-5)

print lo
print up
```

Assign both values to variables

Print both values separately

`sqrt2.py`

$ **python sqrt2.py**
1.4142074585
1.41421508789

But how can we fiddle with pairs of values? How can we get at just one of the values, for example?

The easiest way is to assign a pair of variables to the pair of values in a single operation, as shown. This is analogous to how we assigned a list of variables to a list of values; we can assign a pair of variables to a pair of values.

(Note that when we `print` out the values separately, as here, `print` "prettifies" them, whilst when we `print` the pair of values as we did before, `print` leaves the individual values in the pair alone and displays them "as is".)

Let's break for an exercise…

Write a function that takes a list of numbers as input, and returns the following:

• smallest number in list
• arithmetic mean of list
• largest number in list

If you run into problems with this exercise, ask the course giver or a demonstrator for help.

108

So let's break for another exercise.

Write a function that takes a list of numbers as its input and returns the smallest number in the list, the arithmetic mean (also known as the *average*) of the numbers in the list, and the largest number in the list. You may assume that the list has at least one number in it.

You should test your function after you've written it to make sure it works properly.

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

Also, if you have any questions on anything we've done so far, now would be a good time to ask them.

This page intentionally left ~~blank~~ silly

The exercise just set is fairly straightforward, but if, and only if, you are not getting anywhere with it, then turn the page and have a look at the answer. Then tell the course giver where you were having difficulty with the exercise.

…and regardless of how easy or difficult you are finding the exercise, pause for a moment and reflect on the genius that is Monty Python's Flying Circus.

```
def stats(numbers):                    Answer

    min = numbers[0]
    max = numbers[0]
    total = 0

    for number in numbers:
        if number < min:
            min = number
        if number > max:
            max = number
        total = total + number

    return (min,
            total/(len(numbers)+0.0),
            max)

                            utils.py
```

n.b. Function *fails* if the list is empty.

Here's my answer to the exercise set over the break. Note that the function fails if the list is empty. (I've called my function `stats()`, but you can of course call your function whatever you want.)

Note also that I initially set `total` to `0` rather than `0.0`. This is because I didn't specify whether the numbers in the list that the function takes as input were integers or floating point numbers (or complex numbers for that matter). If I set `total` to the *integer* `0`, then as soon as I add one of the numbers from the list to it, it will be converted (coerced) to the correct type. If, however, I were to set `total` to `0.0` and my function was given a list of integers, then all those integers would be turned into floating point numbers before being added to `total`, potentially unnecessarily losing precision.

Finally, note that when I do the division required to calculate the arithmetic mean (or average) from `total`, I force one of the numbers to be a floating point number (by adding `0.0` to it) to ensure that we don't inadvertently do integer division.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

Tuples

(42 , 1.95 , 'Bob')

(-1 , +1)

('Intro. to Python', 25, 'TTR1')

"not the same as lists"

111

So what are these strange bracketed collections of values we've been using for returning multiple outputs from our functions?

These collections of values (pairs, triplets, quads etc.) are collectively called "tuples" and are the next data type we will look at.

Note that technically tuples don't have to be surrounded by parentheses (round brackets), although they almost always are. A list of comma separated items not surrounded by *any* brackets is also a tuple. For instance, both of the collections of values below are tuples, although only one is explicitly surrounded by round brackets:

```
>>> ('Bob', 1.95, 42)
('Bob', 1.95, 42)
>>> 'Bob', 1.95, 42
('Bob', 1.95, 42)
```

You'll note that when Python evaluates the second tuple it displays the round brackets for us.

Tuples are not the same as lists

(minimum, maximum)

(age, name, height)

(age, height, name)

(age, height, name, weight)

Independent, grouped items

Related, sequential items

[2, 3, 5, 7]

[2, 3, 5, 7, 11]

[2, 3, 5, 7, 11, 13]

The first question has to be "what is the difference between a tuple and a list?"

The key difference is that a tuple has a fixed number of items in it, defined by the script and that number can never change. A list, on the other hand, is a flexible object which can grow or shrink as the script develops. Our root finding function, for example, takes four values as its arguments, not an open-ended list of them. It returns precisely two values as a pair.

If you just want to bundle up a fixed collection of values to pass around the program as a single object then you should use a tuple.

For example, you might want to talk about the minimum and maximum possible values of a range. That's a pair (minimum, maximum), not a list. After all, what would the third item be in such a list?

If you are dealing with statistics of people you might want to bundle name, height in metres and age in years. That's a triplet (3-tuple) of a string, floating point number and integer. You might add a fourth component later (weight in kilograms) but it doesn't follow as the result of a sequence. There is no ordering in these components. Your program would make as much sense if it worked with (age, name, height) as if it worked with (height, age, name).

A set of primes would not be a good tuple. There is an obvious ordering. The list `[2, 3, 5, 7, 11]` makes more sense than the list `[3, 7, 2, 11, 5]`. There is also an obvious "next item": 13.

There is no constraint on the types of the components of a tuple. Because it is fixed and there is no concept of "next component" the types can be safely mixed. A triplet could have one integer, one floating point number and one string (age in years, height in metres, and name say) with no difficulties.

Access to components

Same access syntax as for lists:

>>> **('Bob', 42, 1.95)[0]**
'Bob'

But tuples are *immutable*:

>>> **('Bob', 42, 1.95)[1] = 43**
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment

Another way to consider a tuple is that you usually access all the values at once. With a list you typically step through them sequentially.

It is possible to access individual elements of a tuple and the syntax is unfortunately very similar to that for lists. It is almost always a bad idea to access individual elements like this; it defeats the purpose behind a tuple.

```
>>> person = ('Bob', 1.95, 42)
>>> person
('Bob', 1.95, 42)
>>> name = person[0]
>>> name
'Bob'
```

This is a bad idea. You are much better off doing this:

```
>>> (name, height, age) = person
>>> name
'Bob'
```

If you really don't want `age` and `height`, you can always `del` them.

The third difference is that items in a list can be changed; components of a tuple can't be. Just as the components of a tuple are accessed all at once, they can only be changed all at once, basically by replacing the whole tuple with a second one with updated values.

If we try to change the value of a single component:

```
>>> person = ('Bob', 1.95, 42)
>>> person[2]
42
>>> person[2] = 43
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

This is another reason not to dabble with direct access to individual components of tuples.

113

String substitution


```
>>> 'I am %f metres tall.' % 1.95
'I am 1.950000 metres tall.'
```

Substitution operator

Probably not what we wanted

We started with tuples as a way to get multiple values in and out of functions.

Then we looked at them as items in their own right, as means to collect values together. (You can see another example of this use of tuples in the file `chemicals3.py` in your course home directories which we will use in an exercise in a little while.)

The third use we can make of tuples involves "string substitution" (also known as "string formatting"). This is a mechanism for a (typically long) string to have values (from a tuple) substituted into it.

Python uses the "%" operator to combine a string with a value or a tuple of values.

On the slide above, inside the string a "%f" marks the point where a floating point number needs to be inserted.
Note that the formatting might not be all you wanted.

Substituting multiple values

>>> **'I am %f metres tall and my name is %s.'**

%   (1.95, 'Bob')

'I am 1.950000 metres tall and my name is Bob.'

Note that if you only want to substitute a single value into a string you don't need to use a tuple (although you can if you wish). However, to substitute multiple values you need to use a tuple. The number of markers in the string must match the number of items in the tuple exactly.

There are a number of these substitution markers:
%d   integer
%f   floating point number
%s   string

The complete list of these markers can be found in the Python documentation at the following URL:
http://docs.python.org/library/stdtypes.html#string-formatting-operations

(For those of you familiar with C/C++, the substitution markers used by Python for formatting are very similar to the format specifiers used by C's sprintf() function. And if you're not familiar with C or C++, then just ignore that remark.)

Formatted substitution

standard float marker

```
>>> '%f'    %    0.23
'0.230000'
```
six decimal places

modified float marker: ".3"

```
>>> '%.3f'  %    0.23
'0.230'
```
three decimal places

We can go further. It is possible to specify formatting in the marker to specify exactly how a float, for example, should be rendered. If we interpose a ".3" between the "%" and the "f" then we get three decimal places shown. (The default for the %f marker is to display the floating point number to six decimal places.)

Note that you can use the substitution markers to convert your values into different types as the value is substituted into the string. For example, using %d with a floating point number would cause the floating point number to be substituted into the string as an integer, with the corresponding loss of precision:

```
>>> '%d' % 1.98
'1'
```

More complex formatting possible

```
'23'      '23.4567'    '   23.46'
'   23'   '23.456700'  '23.46   '
'0023'    '23.46'      '  +23.46'
'  +23'   '+23.4567'   '+23.46  '
'+023'    '+23.456700'
'23   '   '+23.46'        'Bob'
'+23  '   '0023.46'       'Bob   '
          '+023.46'       '   Bob'
```

There is a very wide variety of formatting. A guide to it is appended to the notes but we're not going to tediously plough through it all here.

Uses of tuples

1. Functions

2. Related data

3. String substitution

So, as we have seen, there are three main uses for tuples:

- **Functions:** for holding the input and output of functions;
- **Related data:** for grouping together related data that doesn't form a natural sequence and/or is of differing types; and
- **String substitution:** for holding the values that will be substituted into the string.

They are not to be confused with lists, which although similar, are not the same. Lists are to be used for sequential data, where that data is all of the same type.

```python
#!/usr/bin/python

# The keys of this dictionary are the
#  symbols for the atomic elements.
# The values are tuples:
#  (name, atomic number, boiling point).
chemicals = {…}

# For each key in the chemicals
#  dictionary, print the name and
#  boiling point (to 1 decimal place),
#  e.g. for the key 'H', print:
#          hydrogen: 20.3K
```

What goes here?

chemicals3.py

Time for another exercise.

In your course home directories you will find an incomplete script called `chemicals3.py`.

Complete this script so that for each key in the `chemicals` dictionary, the script prints out the name of the atomic element and its boiling point in Kelvin, formatted as shown on the slide above.

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

Also, if you have any questions on anything we've done so far, now would be a good time to ask them.

(By the way, the authors know that the dictionary really contains data about the *atomic elements* rather than chemicals, and so it would be better to call this dictionary "`elements`" rather than "`chemicals`". However, people talk of "elements of lists" or "elements of dictionaries" to refer to the individual items in lists or dictionaries and we would rather not cause unnecessary confusion.)

This page intentionally left blank

Deze bladzijde werd met opzet blanco gelaten.

このページは計画的にブランクを残ている

Ta strona jest celowo pusta.

Esta página ha sido expresamente dejada en blanco.

Эта страница нарочно оставлена пустой.

Denne side med vilje efterladt tom.

Paĝon intence vaka.

این صفحه خالي اسـت

An leathanach seo fágtha folamh in aon turas.

120

This page intentionally left blank: nothing to see here.   If you're stuck for an answer to the exercise, have a look at the next page.

```python
#!/usr/bin/python

# The keys of this dictionary are the
#  symbols for the atomic elements.
# The values are tuples:
#  (name, atomic number, boiling point).
chemicals = {…}

# For each key in the chemicals
#  dictionary, print the name and
#  boiling point (to 1 decimal place),
#  e.g. for the key 'H', print:
#         hydrogen: 20.3K
for symbol in chemicals:
    (name, number, boil) = chemicals[symbol]
    print "%s: %.1fK" % (name, boil)
del name, number, boil
del symbol
```

chemicals3.py

121

And here's one possible solution to the exercise you were just set. Note how I access all the items in the tuple at once, and then only use the ones I actually want.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

(As already mentioned, the authors know that the dictionary in this example really contains data about the *atomic elements* rather than chemicals, and so it would be better to call this dictionary "`elements`" rather than "`chemicals`". However, people talk of "elements of lists" or "elements of dictionaries" to refer to the individual items in lists or dictionaries and we would rather not cause unnecessary confusion.)
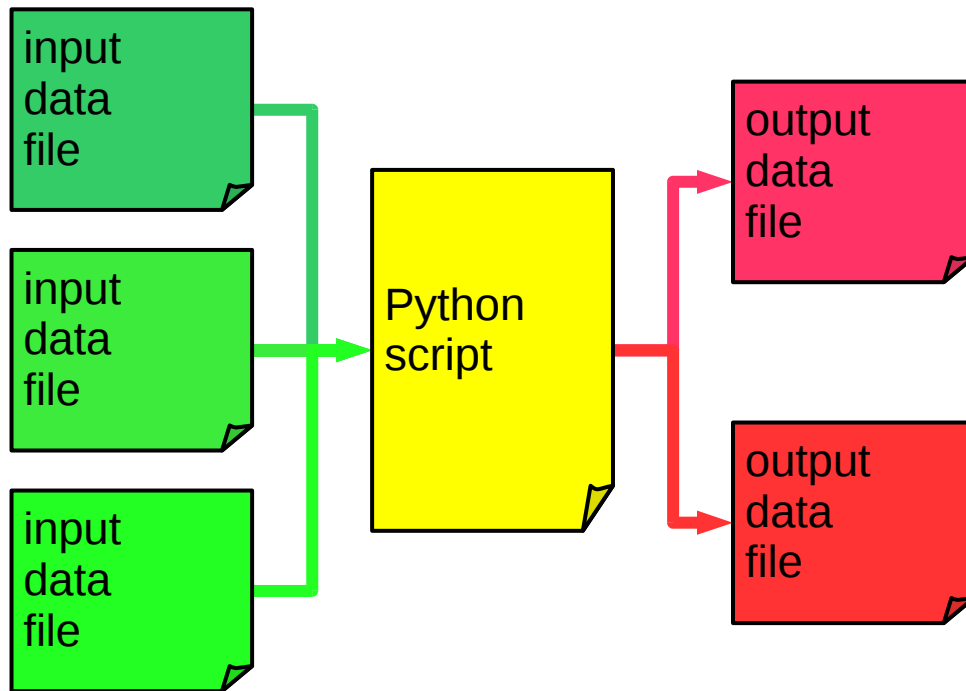
Accessing the system

1. Files

2. Standard input & output

3. The command line

Next we're going to look now at three aspects of how a Python script can access the system.

First we will consider access to files and then move on to how files redirected into and out of the script work. Finally we will examine access to the command line itself.

May want to access many files

The usual Unix approach to input and output is to redirect files into and out of the script using the shell's redirection operations. We will move on to this model shortly but first we want to handle files directly.

In particular, we want to be able to cope with the situation where there is more than one input and/or output file, a situation which *can* be dealt with using shell redirection but which stretches its flexibility.
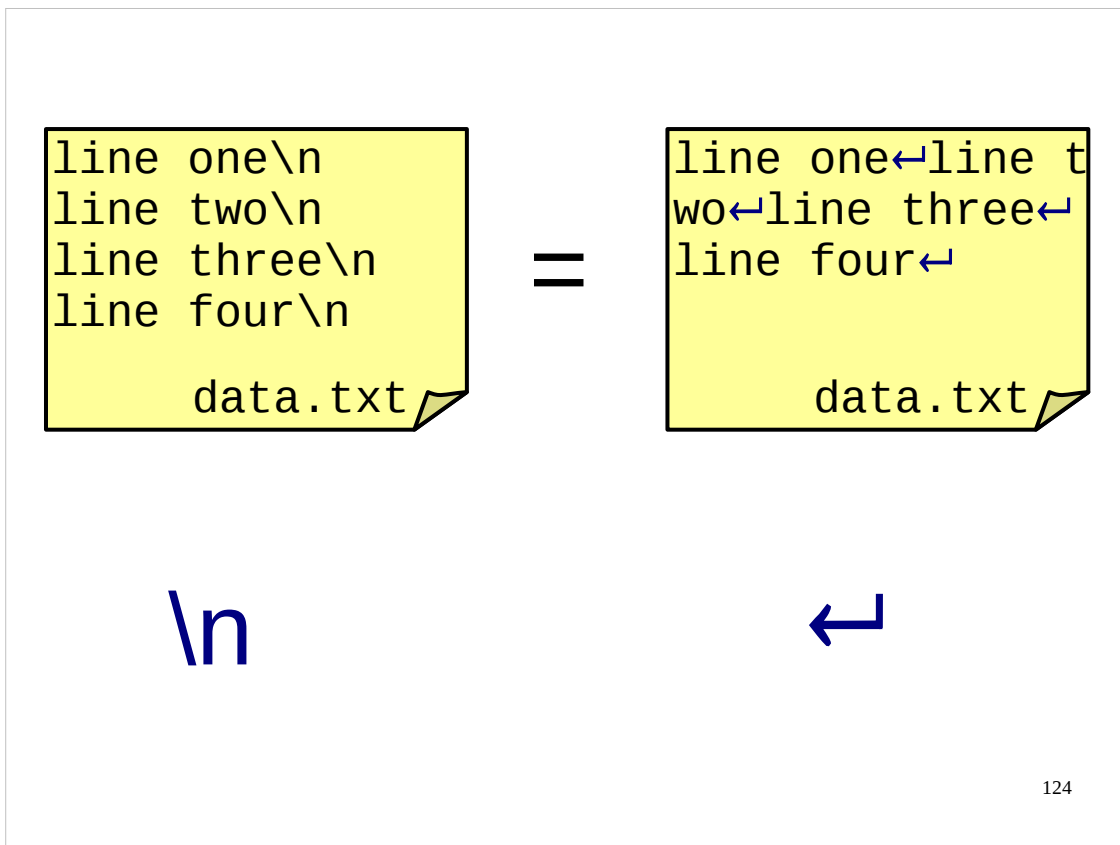
There is a very important point to note about reading input from a file in Python. The input passed to Python consists of *strings*, regardless of any intended meaning of the data in the file. If we supply Python with a file that we know contains numerical values, Python doesn't care. It reads in a series of strings which just so happen to only use a few of the usual characters (digits, decimal points, minus signs and new lines). Python can't be told "this is a file of numbers"; it only reads in strings.

In keeping with its string fixation for file input, Python expects all the files it reads to consist of *lines*. As far as Python is concerned, a line is a string that ends with an "end of line" (EOL) marker (usually the "new line" character, conventionally represented as "\n"). Python can cope with operating systems that use different EOL markers, but we won't cover that in this course. (This is covered in the "Python: Further Topics" course, details of which are available at:

http://training.csx.cam.ac.uk/course/pythonfurther

)

When Python reads a line from a file it will return the string it has read, complete with the "new line" character at the end. If we don't want this trailing "new line" character at the end of our string, we need to get rid of it ourselves. We'll see how to do this shortly.

When you create a file in a text editor, you move to a new line by just pressing the return or enter key on your keyboard. When the file is written to disk these "line breaks" are stored as the appropriate EOL marker for your operating system (under Unix, this is the "new line" character, "\n").

In your course home directories there is a file called "data.txt" which we'll be working with as we investigate how Python reads files. The contents of this file are shown on the slide above.

124
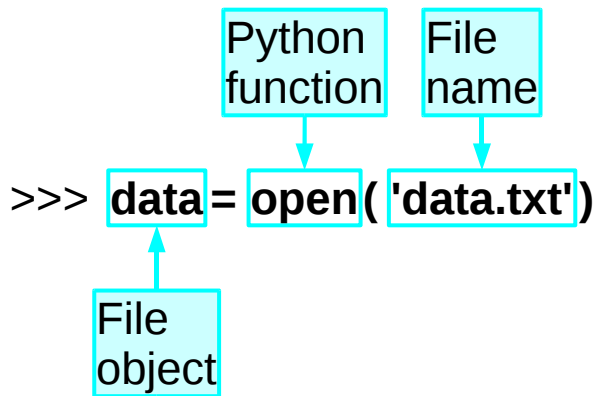
Opening a file involves taking the file name and getting some Python object whose internals need not concern us; it's another Python type, called "`file`" logically enough. If there is no file of the name given, or if we don't have permission to get at this file, then the opening operation fails. If it succeeds we get a `file` object that has two properties of interest to us. It knows the file on disc that it refers to, obviously. But it also knows how far into the file we have read so far. This property, known as the "offset", obviously starts at the beginning of the file when it is first opened. As we start to read from the file the offset will change.

We open a file in Python with the "`open`" command.

In your course home directories there is a file called "`data.txt`". If you enter Python interactively and give the command
```
>>> data = open('data.txt')
```
then you should get a `file` object corresponding to this file inside Python.

Note that we just gave the name of the file, we didn't say where it was. If we don't give a path to the file then Python will look in the current directory. If we want to open a file in some other directory then we need to give the path as well as the name of the file to the `open` command. For instance, if we wanted to open a file called "`data.txt`" in the `/tmp` directory, we would use the `open` command like this: `open('/tmp/data.txt')`.

If you want to know which directory is your current directory, you can use a function called `getcwd()` ("**get current working directory**") that lives in the `os` module:
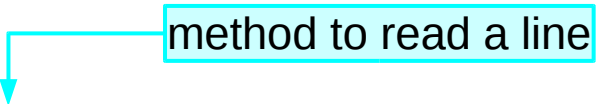```
>>> import os
>>> os.getcwd()
'/home/x241'
```
(If you try this on the computer in front of you, you will find that it displays a different directory to the one shown in these notes.)

You can change your current directory by using the `chdir()` ("**change directory**") function, which also lives in the `os` module. You give the `chdir()` function a single argument: the name of the directory you want to change to, e.g. to change to the `/tmp` directory you would use `os.chdir('/tmp')`. However, **don't try this now**, as if you change the current directory to something other than your course home directory then many of the examples in this section of these notes will no longer work! (If you have foolishly ignored my warning and changed directory, and don't remember what your course home directory was called (and so can't change back to it), the easiest thing to do is to quit the Python interpreter and then restart it.)

```
>>> data = open( 'data.txt')

                              method to read a line

>>> data . readline()

'line one\n'                  first line of file,
                              complete with "\n"


>>> data.readline()          same command

'line two\n'                 second line of file


                                                  126
```

To read a file line by line (which is typical for text files), the `file` object provides a method to read a single line. (Recall that methods are the "built in" functions that objects can have.) The method is called "`readline()`" and the `readline()` method on the `data` object is run by asking for "`data.readline()`" with the object name and method name separated by a dot.

There are two important things to notice about the string returned. The first is that it's precisely that: one line, and the first line of the file at that. The second point is that, as previously mentioned, it comes with the trailing "new line" character, shown by Python as "\n".

Now observe what happens if we run exactly the same command again. (Python on PWF Linux has a history system. You can just press the up arrow once to get the previous command back again.) This time we get a different string back. It's the second line.

```
>>> data = open( 'data.txt')
>>> data.readline()
'line one\n'
>>> data.readline()
'line two\n'

>>> data.readlines()
[ 'line three\n', 'line four\n' ]  ←  remaining lines
```

There's one other method which is occasionally useful. The "readlines()" method gives all the lines from the current position to the end of the file as a list of strings.

We won't use readlines() much as there is a better way to step through the lines of a file, which we will meet shortly.

Once we have read to the end of the file the position marker points to the end of the file and no more reading is possible (without moving the pointer, which we're not going to discuss in this course).

```
>>> data = open( 'data.txt')
>>> data.readline()
'line one\n'
>>> data.readline()
'line two\n'
>>> data.readlines()
[ 'line three\n', 'line four\n' ]


>>> data.close()          disconnect
>>> del data              delete the variable
```

The method to close a file is, naturally, "close()".

It's only at this point that we declare to the underlying operating system (Linux in this case) that we are finished with the file. On operating systems that lock down files while someone is reading them, it is only at this point that someone else can access the file.

Closing files when we are done with them is important, and even more so when we come to examine writing to them.

We should practice good Python variable hygiene and delete the data variable if we aren't going to use it again immediately.

Treating file objects like lists:

for line in data.readlines():
    do stuff

reads the lines all at once

for line in data:
    do stuff

reads the lines as needed

129

We have seen before that some Python objects have the property that if you treat them like a list they act like a particular list. `file` objects act like the list of lines in this case, but be warned that as you run through the lines you are running the offset position forward.

Very primitive input

line.split() → Very simplistic splitting

→ No way to quote strings
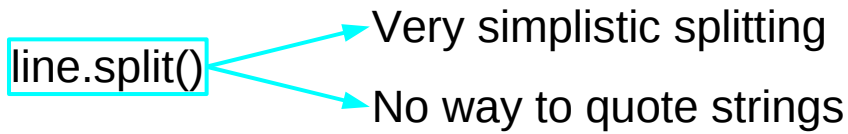
Comma separated values: **csv** module
Regular expressions: **re** module

"Python: Further Topics" course

"Python: Regular Expressions" course

We often use the `split()` method of strings to process the line we've just read in from a file. Note, though, that the `split()` method is very, very primitive. There are many better approaches, and some of these are covered in the "Python: Further Topics" and "Python: Regular Expressions" courses.
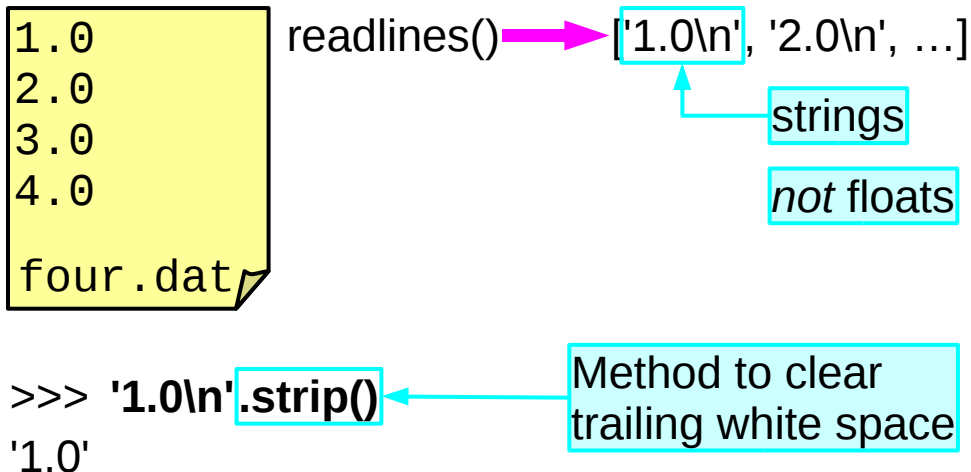
For details of the "Python: Further Topics" course, see:
        http://training.csx.cam.ac.uk/course/pythonfurther

…and for details of the "Python: Regular Expressions" course, see:
        http://training.csx.cam.ac.uk/course/pythonregexp

Reading data gets you strings

```
1.0
2.0
3.0
4.0

four.dat
```

readlines() ➡ ['1.0\n', '2.0\n', …]

strings

not floats

>>> **'1.0\n'.strip()**

'1.0'

Method to clear trailing white space

Still need to convert string to other types

As mentioned before, input from a file is read by Python as a string, complete with a trailing "new line" character.

One method of getting rid of unwanted "white space" (spaces, tabs, "new line" characters, etc.) is to use the `strip()` method of strings. `strip()` returns a copy of a string with all leading and trailing "white space" removed. This is often useful when dealing with strings read in from a file.

```
>>> '1.0\n'.strip()
'1.0'
>>> '   1.0  \n'.strip()
'1.0'
```

Converting from one type to another

In and out of strings

>>> float('0.25')		**>>> str(0.25)**
0.25	↔	'0.25'
>>> int('123')		**>>> str(123)**
123	↔	'123'

We need to be able to convert from the strings we read in to the numbers that we want. Python has some basic functions to do exactly this. Each is named after the type it generates and converts strings to the corresponding values. So `float()` converts strings to floating point numbers and `int()` converts strings to integers. Similarly, the `str()` function converts values into their string representations. The `float()` and `int()` functions will also strip any leading or trailing white space characters (spaces, tabs or new lines) from the string before converting it, which is very useful when working with numbers that were read in as strings from a file.

Converting from one type to another

Between numeric types

>>> **int(12.3)** loss of precision
12

>>> **float(12)**
12.0

The functions are slightly more powerful than just converting between strings and numeric types. They attempt to convert any input to the corresponding type so can be used to convert between integers and floating point numbers, and between floating point numbers and integers (truncating the fractional part in the process).

Converting from one type to another

If you treat it like a list…

>>> **list('abcd')**
['a', 'b', 'c', 'd']

>>> **list(data)**
['line one\n', 'line two\n', 'line three\n', 'line four\n']

>>> **list({'H':'hydrogen', 'He':'helium'})**
['H', 'He']

Even more impressive is the `list()` function which converts things to lists. We have repeated the mantra several times that where Python expects a list it will treat an object like a list. So `file` objects are treated as lists of lines, dictionaries are treated as lists of keys and strings can even be treated as lists of characters. The `list()` function makes this explicit and returns that list as its result.

```
#!/usr/bin/python

# This script reads in some
#  numbers from the file 'numbers.txt'.
# It then prints out the smallest
#  number, the arithmetic mean of
#  the numbers, and the largest
#  number.
```

What goes here? (Use the function you wrote in an earlier exercise.)

135

Time for another exercise.

Write a script that reads in some numbers from the file "numbers.txt" in your course home directories. (It will obviously need to convert the strings it has read in into the appropriate numeric types.)

It should then print out the smallest number, the arithmetic mean (average) of the numbers, and the largest number. (You should use the function you wrote in one of the earlier exercises to do this.)

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

Also, if you have any questions on anything we've done so far, now would be a good time to ask them.

This page intentionally left blank

Deze bladzijde werd met opzet blanco gelaten.

このページは計画的にブランクを残ている

Ta strona jest celowo pusta.

Esta página ha sido expresamente dejada en blanco.

Эта страница нарочно оставлена пустой.

Denne side med vilje efterladt tom.

Paĝon intence vaka.

اين صفحه خالي است

An leathanach seo fágtha folamh in aon turas.

136

This page intentionally left blank: nothing to see here. If you're stuck for an answer to the exercise, have a look at the next page.

```
#!/usr/bin/python

# This script reads in some
#  numbers from the file 'numbers.txt'.
# It then prints out the smallest
#  number, the arithmetic mean of
#  the numbers, and the largest
#  number.

import utils

data = open('numbers.txt')

numbers = []
for line in data:
        numbers.append(float(line))
del line

data.close()
del data

print utils.stats(numbers)
```

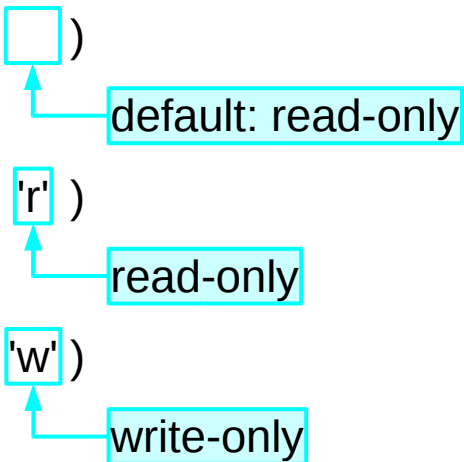**Answer**

function you wrote
in earlier exercise

137

Here's my answer to the exercise. Note that I'm using the `stats()` function I wrote in one of the earlier exercises, which I `def`ined in my `utils` module – you should use the name of whatever function you created as your answer to the earlier exercise.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

Output to files

```
input   =  open('input.dat'    )
```
default: read-only

```
input   =  open('input.dat',  'r' )
```
read-only

```
output  =  open('output.dat', 'w' )
```
write-only

To date we have been only reading from files. What happens if we want to write to them?

The `open()` function we have been using actually takes two arguments. The second specifies whether we want to read or write the file. If it is missing then the default is to open the file for reading only.

(We haven't described how to write functions with optional arguments, nor are we going to in this course – this is covered in the "Python: Further Topics" course; for details of this course see:

    http://training.csx.cam.ac.uk/course/pythonfurther

)

The explicit value you need to open a file for **r**eading is the single letter string `'r'`. That's the default value that the system uses. The value we need to use to open a file for **w**riting is `'w'`.

Output to files

```
>>> output = open('output.dat', 'w')
>>> output.write('alpha\n')          explicit "\n"
>>> output.write('bet')              write(): writes
                                     lumps of data
>>> output.write('a\n')
>>> output.writelines([ 'gamma\n', 'delta\n'])
>>> output.close()                   Flushes to
                                     file system
```

As ever, a newly opened file has its position pointer ("offset") pointing to the start of the file. This time, however, the file is empty. **If the file previously had content then it gets completely replaced.**

Apart from the explicit second argument, the open() function is used exactly as we did before.

Now that we've written our file ready to be written to we had better write something to it. There is no "writeline()" equivalent to readline(). What there is is a method "write()" which might be thought of as "writelump()". It will write into the file whatever string it is given whether or not that happens to be a line.
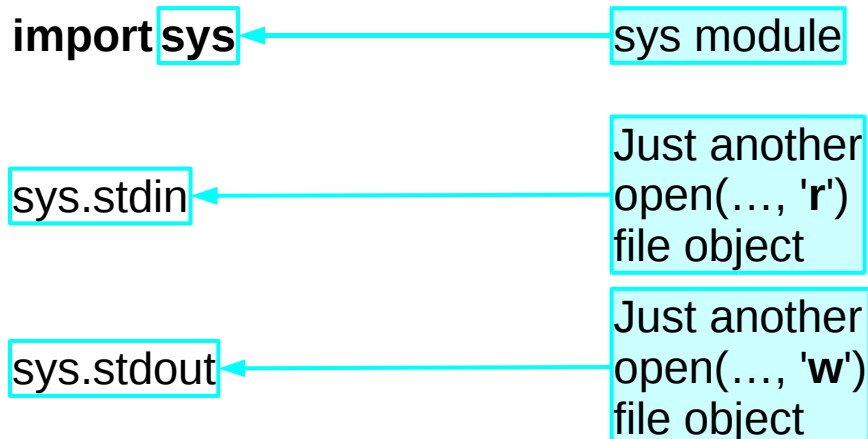
When we are writing text files it tends to be used to write a line at a time, but this is not a requirement.

There is a writing equivalent of readlines() too: "writelines()". Again,the items in the list to be written do not need to be whole lines.

Closing the file is particularly important with files opened for writing. As an optimisation, the operating system does not write data directly to disc because lots of small writes are very inefficient and this slows down the whole process. When a file is closed, however, any pending data is "flushed" to the file on disc. This makes it particularly important that files opened for writing are closed again once finished with.

  It is only when a file is closed that the writes to it are committed to the file system.

Standard input and output

import **sys** ←——————————— sys module

sys.stdin ←——————————— Just another open(…, '**r**') file object

sys.stdout ←——————————— Just another open(…, '**w**') file object

Let's move on to look at a couple of other ways to interface with our scripts. These will involve the use of a particular module, provided on all platforms: "sys", the system module.

First let's quickly recap what we mean by "standard input" and "standard output".

When a Unix command is run and has its input and output set up for it my the shell, e.g.

```
$ command.py < input.dat > output.dat
```
we refer to the data coming from input.dat as the standard input and the data going to output.dat as the standard output. It is critically important to understand that the shell doesn't just pass in the file names to the command. Instead, the shell does the opening (and closing) of the files and hands over file objects to the command.

(Note that above Unix command line is just an example command line we might use, it will not work if you actually try typing it in on the machines in front of you.)

So, what do standard input and output look like inside the Python system?

The sys module gives access to two objects called "sys.stdin" and "sys.stdout". These are file objects just as we got from the open() command. These come pre-opened, though, and will be closed for us automatically when the script ends.

So, what does this script do?

Read lines in from standard input

Write them out again to standard output

It copies files, line by line

```
#!/usr/bin/python

import sys

for line in sys.stdin:
    sys.stdout.write(line)
```

stdin-stdout.py

141

So let's look at a very simple script. It imports the `sys` module and runs through the lines of `sys.stdin`, the standard input. (Recall that if you treat a file object like a list you get the list of lines.) For each line it simply writes that line to `sys.stdout`, the standard output.

In essence it is a copier. Every line it reads in it writes out.

Obviously in practice we would do something more interesting with "`line`" between reading it in and writing it out. This functionality should be carved off into a function.

One approach is for the function to run a test on the line and return a Boolean according to whether or not the line passed. Then the line either is or isn't written out according to the answer.

```
for line in sys.stdin:
    if test(line):
        sys.stdout.write(line)
```

The other approach is for the function to transform the line in some fashion and the script then writes out the transformed line.

```
for line in sys.stdin:
    sys.stdout.write(transform(line))
```

Or we can combine them:

```
for line in sys.stdin:
    if test(line):
        sys.stdout.write(transform(line))
```
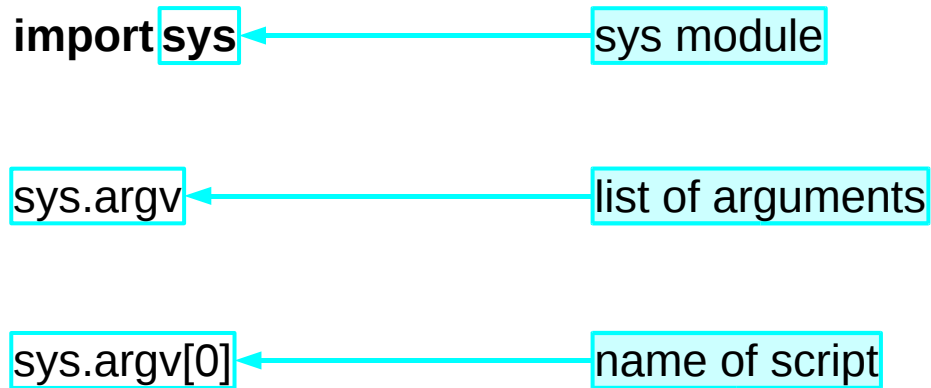
or:

```
for line in sys.stdin:
    newline = transform(line)
    if test(newline):
        sys.stdout.write(newline)
```

As you may know, this model of reading in lines, possibly changing them and then possibly printing them out is called "filtering" and such scripts are often called "filters". Filters are covered in more detail in the "Python: Regular Expressions" course. For details of this course, see:

    http://training.csx.cam.ac.uk/course/pythonregexp

Command line

import `sys` ← `sys module`

`sys.argv` ← `list of arguments`

`sys.argv[0]` ← `name of script`

The next interaction with the system is to get at the command line. To date our interaction with the user has been with files, either opened explicitly inside the script or passed pre-opened by the calling shell. We want to exchange this now to allow the user to interact via arguments on the command line.

There's another object given to us by the `sys` module called "`argv`", which stands for "**arg**ument **v**alues".

Item number zero in `sys.argv`, i.e. `sys.argv[0]` is the name of the script itself.

Item number one is the first command line argument (if any).

Item number two is the second command line argument (if any), and so on.

Note that all the items in `sys.argv` are *strings*, regardless of whether the command line arguments are meant to be interpreted as strings or as numbers or whatever. If the command line argument is not intended to be a string, then you need to convert it to the appropriate type.

```
#!/usr/bin/python

print sys.argv[0]

print sys.argv
```

args.py

$ **python args.py 0.25 10**

args.py

['args.py', '0.25', '10']

NB: list of *strings*

There is a script called `args.py` in your course home directories. You can use this to investigate what the command line arguments of your script look like to Python.

Again, the most important thing to note is that Python stores these arguments as a list of *strings*.

```
#!/usr/bin/python

# This script takes some numbers as
#  arguments on the command line.
# It then prints out the smallest
#  number, the arithmetic mean of
#  the numbers, and the largest
#  number.
```

What goes here?

Time for another exercise.

Write a script that takes some numbers as command line arguments. (Your script will obviously need to do some string conversion.)

It should then print out the smallest number, the arithmetic mean (average) of the numbers, and the largest number. (You should use the function you wrote in one of the earlier exercises to do this.)

Make sure you test your script.

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

Also, if you have any questions on anything we've done so far, now would be a good time to ask them.

```
#!/usr/bin/python

# This script takes some numbers as
#  arguments on the command line.
# It then prints out the smallest
#  number, the arithmetic mean of
#  the numbers, and the largest
#  number.

import sys
import utils

numbers=[]
for arg in sys.argv[1:]:
        numbers.append(float(arg))
del arg

print utils.stats(numbers)
```

Answer

function you wrote earlier

145

Here's my answer to the exercise. Note that I'm using the `stats()` function I wrote in one of the earlier exercises, which I `def`ined in my `utils` module – you should use the name of whatever function you created as your answer to the earlier exercise.

If there is anything in the above solution you don't understand, or if your solution was radically different to the above, please let the course presenter know now.

```
def find_root(
    …
    ):
    """find_root(function, lower, upper, tolerance)
finds a root within a given interval to within a
specified tolerance.  The function must take
values with opposite signs at the interval's ends."""

    while upper - lower < tolerance:
        middle = (lower + upper) / 2.0
        if function(middle)*function(upper) > 0.0:
            upper = middle
        else:
            lower = middle

    return (lower, upper)
```

Inserted string

utils.py

We are going to end with a little frill that is enormously useful. We are going to add some extra documentation to our functions and modules which they can then carry around with them and make accessible to their users.

We edit `utils.py` to insert some text, as a string, immediately after the "`def`" line and before any actual code (the body of the function). This string is long so is typically enclosed in triple double quotes, but this isn't required; any string will do. The text we include is documentation for the *user*. This is not the same as the comments in the code which are for the programmer.

A string placed here does not affect the behaviour of the function.

Doc strings for functions

>>> **import utils**

module

function

doc string

>>> **print utils.find_root.__doc__**

Double underscores

find_root(function, lower, upper, tolerance)
finds a root within a given interval to within a
specified tolerance. The function must take
values with opposite signs at the interval's ends.

So how do we get at it?

If we import the module containing the function we can print the "__doc__" attribute of the module's function. (Just as methods are built in *functions* of Python objects, so "attributes" are *variables* that are built in to Python objects. We use the same dot (.) syntax that we've used for methods to get at attributes of objects.)

The "__doc__" attribute of the function gives us access to the string we added to the function's definition. Ultimately, we will be sharing the modules with other people. This is how we can share the documentation in the same file.

Note that it is a ***double underscore*** on each side of "__doc__". This isn't clear in all fonts.

All the system modules come copiously documented:

```
>>> import sys
>>> print sys.exit.__doc__
exit([status])

Exit the interpreter by raising SystemExit(status).
If the status is omitted or None, it defaults to zero (i.e., success).
If the status is numeric, it will be used as the system exit status.
If it is another kind of object, it will be printed and the system
exit status will be one (i.e., failure).
```

If you need to work with a module, the doc strings are a good place to look first.

Doc strings for modules

String at *start* of file

```
"""A collection of my useful little functions."""

def find_root(
    …
```

utils.py

If we can document a function, can we document a module? Yes.

We insert a similar string at the top of the module file, before any Python code.

Doc strings for modules

>>> **import utils**

module

doc string

>>> **print utils.__doc__**
A collection of my useful little functions.

149

We get at it in the exact same way we got at a function's doc string, except that this time no function name is involved.

```
>>> import sys
>>> print sys.__doc__
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the
interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
…
```

All system modules have doc strings describing their purpose and contents.

Final exercise

```python
#!/usr/bin/python

# This script takes some atomic symbols
#  on the command line.  For each symbol,
#  it prints the atomic element's name, and
#  boiling point (to 2 decimal places),
#  e.g. for the symbol 'H', print:
#  hydrogen has a boiling point of 20.3K
# Finally, it tells you which of the given
#  atomic elements has the lowest atomic
#  number.

# The keys of this dictionary are the
#  symbols for the atomic elements.
# The values are tuples:
#  (name, atomic number, boiling point)
chemicals = {…}
```

chemicals4.py

150

We'll leave you with one final exercise and some references for further information about Python.

In your course home directories you will find an incomplete script called `chemicals4.py`.

Complete this script so that it accepts the symbols for atomic elements on the command line. For each symbol it gets from the command line, it should print out the name of the atomic element and its boiling point in Kelvin to 2 decimal places in the manner given in the example on the slide above.

It should then tell you which of the specified atomic elements has the lowest atomic number.

Make sure you test your script.

If you have any problems with this exercise, please ask the course giver or a demonstrator for help.

You can take a copy of the `chemicals4.py` file away with you and work on it at your leisure, but that does mean that we won't be able to help you should you run into any problems.

Finally, if you have any questions on anything we've covered in the course, now is your last opportunity to ask them.

(And again, for the pedants reading, the authors know that the `chemicals` dictionary in this script really contains the atomic elements rather than chemicals, and so it would be better to call this dictionary "`elements`" rather than "`chemicals`". However, as we said, we want to avoid confusion with the use of the word "elements" in "elements of lists" or "elements of dictionaries" to refer to the individual items in lists or dictionaries.)

References and further courses

Dive Into Python
Mark Pilgrim
Apress
ISBN: 1-59059-356-1
http://diveintopython.org/

Best book on Python your course presenter has found. (It was written for Python 2.3, though. Luckily, Python 2.4, 2.5 and 2.6 are very similar to Python 2.3.)

Official Python documentation: http://docs.python.org/

"**Python: Further Topics**" is the follow-on course from this one. For details of this and other University Computing Service courses on Python, see:

http://training.csx.cam.ac.uk/theme/scicomp?scheduled=all

151

If you need to learn more Python than that covered in this course, have a look at the other Python courses in the "Scientific Computing" series:

```
http://training.csx.cam.ac.uk/theme/scicomp?scheduled=all
```

The course which directly follows on from this course is the "**Python: Further Topics**" course:

```
http://training.csx.cam.ac.uk/course/pythonfurther
```

The official Python documentation can be found on-line at:

```
http://docs.python.org/
```

You can either browse it on-line or download it in a variety of formats. It is extremely comprehensive and covers all aspects of the language and all the core modules supplied with Python. It also includes a tutorial on Python.

There are also a number of good books on Python out there that you may wish to consult. Probably the best book on Python your course presenter has found is "**Dive Into Python**" by Mark Pilgrim. It also has the advantage of being available free, in a variety of electronic formats, at:

```
http://diveintopython.org/
```

If, however, you find this book as useful as your course presenter does, then you should really consider buying a paper copy as such excellent free resources deserve our support. The one caveat regarding this book is that when it was written the major release of Python then current was 2.3. At the time of writing these notes, the current major release of Python in widespread use is 2.6. Fortunately, Python 2.4, 2.5 and 2.6 are not wildly different to Python 2.3, so this shouldn't cause too many problems. It does mean, though, that some of the external modules referred to in the book have either been superseded or are no longer maintained/available.