

This is the UCS three afternoon course on Python for people who have no experience of programming at all. We warn all those people who *do* have some programming experience and who are here just to add the Python notch to their bed post that they will be excruciatingly bored in this course. Those people who do already know how to program in another language and want to learn Python are better off attending the UCS "Python: Introduction for Programmers" one day course. For details of this course, see http://training.csx.cam.ac.uk/course/python4progs

Note that the UCS Python courses cover Python 2.4 to 2.6, which are the most common versions currently in use – it does NOT cover the recently released Python 3.0 since that version of Python is so new. In some places Python 3.0 is significantly different to Python 2.x, and this course will be updated to cover it as it becomes more widely used.

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



So what will this course cover?

We will start with a brief introduction to Python, looking briefly at what it is used for and how we launch it on the systems being used for this course.

Once we have it running we will start by using it as a glorified calculator to get us used to its features. We will examine how it handles numbers, text and the concept of a statement being true or false.

Course outline — 2		
Using Python like a programming language	Variables ifthenelse while loops Comments <b>Lists</b> for loops Functions	
We will do <i>lots</i> with lists.	Tuples Modules	
UCS		3

But Python is there for us to use as a programming language so, after spending a while using it as a manually operated calculator, we will start to use it as a fully-fledged programming language.

As part ofd this we will look at how Python stores values and assigns names to these stored values. We will look at the three fundamental constructs that will allow us to build programs that actually do something. ("if...then...else...", "while... loops", and "for... loops")

We will also spend a lot of time looking at how Python handles lists. There are two reasons for this. First, Python uses lists a lot so we need to understand them. Second, Python lists are the first example of a computer data structure that doesn't have any analogue in the usual arithmetics.

Then we will look at writing our own functions that use what we have learnt. Functions permit us to structure our code in a more maintainable fashion. We will look at how Python groups related functions together and what groups of functions is provides ready-made. These groups are called "modules" in Pythonic language.



Once we know the rudiments of programming in Python we will look at the support functions offered by the base Python system. These will let us access the system outside of Python. The main example of this will be accessing the file system. Finally we will look at one last, very powerful mechanism for storing data, the "dictionary".



I want to start by convincing you that learning Python is worthwhile. Python is used for every scale of operation. Here is a spectrum of examples running from the largest to the smallest.

The Massively Multiplayer Online Role-Playing Game (MMORPG) "Eve Online" supports over 300,000 users with a Python back end.

http://wiki.python.org/moin/PyCon2006/Talks#line-196

Two very common frameworks for web applications are Django (general purpose) and Plone (content management). Both are implemented in Python.

www.djangoproject.com plone.org

On the desktop itself there are frameworks to build graphical applications in Python. The two standard Unix desktop environments are called GNOME and Qt. Both have Python support. There is similar support under Windows and MacOS.

www.pygtk.org www.pyside.org www.wxpython.org

There are plenty of command line programs written in Python. Some Unixes (e.g. OpenSUSE) have a helper program they call when the user asks for a command the shell doesn't know. That helper program is written in Python.

Within programs there are support libraries for almost every purpose including a very powerful scientific python library called "SciPy" ("Sigh-Pie") and an underlying numerical library called "NumPy".

www.scipy.org

Python is also used to control instruments (a simple robot is featured in the slide) and is also used in embedded systems. The card shown is ""…IEEE802.15.4 based, auto-forming, multi-hop, instant-on, mesh network stack combined with an embedded Python interpreter for running application code."

synapse-wireless.com



Languages split into two broad camps according to how they are used, though it is better regarded as a spectrum rather than a clean split.

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". Fortran, C and C++ are examples of languages that are treated in this way.

Interpreted languages are stored as the text written by the programmer and this is read by another program, called the interpreter, typically one line t a time. The line is read and parsed by the interpreter which then executes any instructions required itself. Then it moves on to the next line. Note that the interpreter is typically a compiled program itself.

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, often called a "script".

If the interpreter (Python in our case) gets its 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 batch for anything complex.



To launch a terminal window to type commands into launch the GNOME Terminal application from the menu system:

Applications  $\rightarrow$  Unix Shell  $\rightarrow$  GNOME Terminal

In the 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 character (\$). On PWF Linux the prompt is actually that character preceded by some other information.

Our other 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.



At the 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 character (\$). On PWF Linux the prompt is actually that character preceded by some other information.

Our other 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. For completeness what they mean is this:

- 1. The version of Python this is.
- 2. The version of the C compiler the interpreter was compiled with.
- 3. A few hints as to useful commands to run.

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 information that is passed to the function like this is called its "arguments". In our case there is only one argument. Arguments are passed in brackets to group them together.

(Actually, in Python the print function is a special case for historical reasons, and doesn't seed the brackets. However, this special exemption is scheduled for removal in the next version of Python so we encourage you to get in the habit of using them from the start.)

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 or Python keywords.

The command is executed and the text "Hello, world!" is produced. The print command always starts a new line after outputting its text. Note that the quotes were used to indicate to Python that their contents were text but they are not part of the text itself so are not printed out as part of the print command's output.

Once the command is complete the Python interpreter is ready for another command so prompts for it with the same triple chevron ("greater than" sign) marker, ">>>".

Note that everything in Python is case-sensitive: you have to give the print command all in lower-case; "PRINT", "pRiNt", etc. won't work.



We will continue in our use of this interactive python session.

We issue a trivial command:

## >>> print(3)

and Python faithfully prints the number

3

to the terminal.

If, however, we just type a bare number:

>>> 5

then Python evaluates whatever it has been given and also outputs the result of that evaluation:

5

Then Python prompts for more input.

There is a subtle difference in the two behaviours. In the first case we explicitly told Python to print a value. In the second we gave it a value and it responds, essentially saying "yup, that's a 5".



We can take this further. We will meet numbers shortly but note for now that the "evaluation" need not always be trivial. We can use Python to evaluate expressions.

Using Python interactively	
<pre>&gt;&gt;&gt; print('Hello, world!') Hello, world! </pre> No quotes	
>>> <b>'Hello, world!'</b> 'Hello, world!'	
UCS	13

The difference is more explicit if we use text rather than numbers.

In the first case we use the quotes to mark their content as text. When we ask Python to print some text it prints just the text itself without any syntactic markers. So the print example has no quotes in its output.

In the second case we hand this text object to Python and it says "yup, this ia a text object containing this sequence of characters. The way it indicates that it is a text object is by enclosing it in quotes. It uses exactly the same marker as we did.



Now that we know how to get into Python we need to know how to get out of it again. In common with many Unix commands that read input from the keyboard, the program can be quit by indicating "end of input". This is done with a "[Ctrl]+[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 only once. The [Ctrl]+[D] key combination, meaning "end of input" or "end of file", also means this to the underlying Unix command interpreter. If you press [Ctrl]+[D] twice, the first kills off Python returning control to the Unix command line and the second kills that off. 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

>>>

If you do not feel comfortable using [Ctrl]+[D] then you can type run the Python command exit() instead.



Here's a quick exercise. It shouldn't take you too long, but if you get stuck do get the demonstrator's attention and ask.

The answers to 4(a) and 4(b) should come as no surprise. The answers to 4(c) and 4(d) will be new but we will cover them later in this course.

If you accidentally quit your terminal window as well as your Python session then you need more practice with Control characters. Launch another terminal window, launch Python in it and have another go at exiting cleanly.

If you rush through this exercise and are left with 2 minutes 30 seconds of thumbtwiddling time here are some more exercises:

A. Try to predict what each of these interactive Python commands will result in. Then try them for real. Were you right?

>>> 99 - 100 >>> 123456789 + 987654322 >>> 99 > 100

B. The first of these commands works. The second gives an error. Why do you think it fails? (We will address this when we cover text properly later.)

```
>>> print('Dowling')
>>> print('O'Connor')
```



Now we have seen Python interactively (though in a very limited capacity) we should look at it being used in batch mode: on files of Python commands. To read and write these files we will use a simple editor in this course called "gedit". If you already know a different Unix plain text editor you are welcome to use it, but the course notes and the lecturer will use gedit. A hand out is provided with a quick guide on how to use it.

To launch gedit on PWF Linux select Applications  $\rightarrow$  Word and Text Processing  $\rightarrow$  gedit from the menus.

Please be careful. The gedit application edits plain text files. Some of these (and most for our purposes) will be Python scripts, but it has nothing to do with Python itself. It is just a text editor.



So, once we have a script (as we can see in gedit) we need to run it. We do this in the terminal window by running the python command just as we did interactively but this time we add the name of the script file we want it to run.

#### \$ python hello.py

Hello, world!

### \$

Please keep the text editor and the terminal window separate in your mind.



Note that Python runs the command inside the file just as if it had been typed interactively. The only difference is that this time Python does not print the three lines of introductory blurb and exits automatically once the script is complete. We go straight back to the Unix prompt; we do not need to quit from Python ourselves.



We will use this representation of file contents rather than screenshots in future slides. There is another difference between interactive and batch mode which we can see with the script three.py.

## >>> python three.py

## 3

Not only does batch mode drop the introductory blurb but it also drops the output of values. Unless there is an explicit output command, Python in batch mode is silent.



Those are the only two differences between interactive and batch mode Python. Apart from that, it's just a case of what's more convenient.

# Progress

What Python is

Who uses Python

How to run Python interactively

How to run a Python script

UCS

21



Here's an exercise to make sure you can run scripts and also edit them.

Types of values		
Numbers	Whole numbers Decimal numbers	
Text		
"Boolean"	True False	
UCS		23

We are going to start by using Python as a glorified calculator. To do that we need to know a bit about the sorts of things we will be calculating with. We need to know a little about how Python handles its various values.

In computing values get divided up into "types". So The number 3 is not the same as the letter "3". These have different types.

We will start by looking at just a few types. These will be plenty to get us a long way. We will look at numbers, both whole numbers and decimal numbers, we will look at text and we will look at so-called "boolean" values. These are what the Python system uses to record "true" and "false". We will see them in detail shortly.



We will start with the integers, i.e. the "whole numbers" (0, the positive whole numbers and the negative whole numbers) = {..., -3, -2, -1, 0, 1, 2, 3, ...}. The letter  $\mathbb{Z}$  (with the double diagonal stroke) is the mathematical symbol for the whole numbers, known mathematically as the "integers".



If we type "4+2" at the Python prompt it is evaluated and returned as "6". There's no great surprise there. It should be noted that Python doesn't care about spaces or the lack of them around the plus sign, or before or after the integers for that matter.



Subtraction also behaves in a similar fashion with negative numbers represented with a leading minus sign.



We see our first deviation from "obvious" with multiplication. The plus and minus signs appear on the standard keyboard so can be used by programming languages. The times sign, "×", does not appear on the keyboard so traditionally in computing the asterisk, "\*", is used instead. (Actually Linux systems with UK keyboards can get "×" as [1]+[AltGr]+[,].)



Similarly, division uses the forward slash character, "/", rather than " $\div$ ".

Division is the first place where Python's integer arithmetic differs from conventional maths. We are working in integers and Python remains within integers for the results too so if the division would give a fractional answer Python rounds down to give an integer value. So the expression "5/3" gives "1" rather than "1 2/3". Note that the "round down" rules is applied absolutely. As a result "-5/3" is evaluated to be "-2" which is the integer below "-1 2/3". So (-5)/3 does not evaluate to the same as -(5/3). This sort of integer division is also known as "floor division".

(Again, " $\div$ " is [ $\hat{1}$ ]+[AltGr]+[.] on a Linux system with a UK keyboard, if you are interested.)



The next mathematical operator we will describe for integers is raising to powers (this is known as "exponentiation"). In classical arithmetic notation this is represented by the use of superscripts, so "4 to the power of 2" is written " $4^2$ ". However, this cannot be represented on a standard keyboard so instead a different notation is used. We write  $4^2$  as " $4^**2$ ". 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 this operator rather than "\*\*". Python, however, uses "\*\*" for this, and uses "^" for something completely different that will not encounter in this introductory course.



There is one integer operator used in computing which does not have a classical equivalent symbol. The percent character is used to to determine remainders. "5%3" gives the answer "2" because 5 leaves a remainder of 2 when divided by 3. The remainder is always zero or positive, even when the number in front of the percent character is negative.

We won't be using this operator in the course; it is included merely for completeness.

>>> <b>2 * 2</b> 4	How far can integers go?
>>> 4 * 4	
16	
>>> 16 * 16	
256	
>>> 256 * 256	
65536	So far, so good
UCS	31

Python's integer arithmetic is very powerful and there is no limit (except the system's memory capacity) to 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.

>>> 65536 * 65536
4294967296 <mark>L</mark>
>>> 4294967296 * 4294967296
18446744073709551616L
>>> 18446744073709551616 * 18446744073709551616
340282366920938463463374607431768211456L
No limit to size of Python's integers! 32

If we square that 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". 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.

Note: If you are using a system with a 64-bit CPU and operating system then the number just over four billion also comes without an "L" and it kicks in one squaring later.



It is worth mentioning that 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. 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. 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. Square it one more time and Python can still beat them.

Progress	
Whole numbers	2, -1, 0, 1, 2
No support for fractions	1/2 0
Unlimited range of values	S
Mathematical operations	
Maths: a+b a-b ucs Python: a+b a-b	b a×b a÷b a <sup>b</sup> a mod b b a*b a/b a**b a%b <sub>34</sub>

E>	kercise				
In P	ython, calcula	te:			
1. 3. 5. 7. 9.	12+4 12-4 12×4 12÷4 12 <sup>4</sup>	2. 4. 6. 7. 10.	12+5 12-5 12×5 12÷5 12 <sup>5</sup>		
Which of these answers is "wrong"?					
UCS	5			<ul><li>✓ 2 minutes</li></ul>	35

Here are some simple integer sums to do in Python. By "wrong" I mean that the integer answer from Python does not equal the mathematical non-integer answer.



And that wraps it up for integers.

Next we would like to move on to real numbers, i.e. the whole numbers and all the values in between, so that we can cope with divisions that give fractional answers and other more complex mathematical operations that need more than the integers.

Python implements a scheme to represent real numbers called "floating point numbers". Some non-integer numbers can be represented exactly in this scheme. Two examples are  $1\frac{1}{4}$  and  $1\frac{1}{2}$ . Most numbers can't be.

Incidentally, 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.


But what about an equally "simple" fraction, 4/3? In normal mathematical representation we express this approximately as a decimal expansion to a certain number of places. This is the approach computers take, typically specifying the number of decimal places they will work to in advance.

( $\mathbb{R}$  is the mathematical symbol for the real numbers.)

If you are going to be doing numerically intensive work you 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

This article will tell you more about the downright weird behaviour of floating point numbers and the kinds of problems this can cause in your programs. Note, however, 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.



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.) 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.

>>> <b>0.1</b> 0.1	1/10 is stored inaccurately.
>>> <b>0.1 + 0.1 + 0.1</b> 0.30000000000000000004	
Floating point numbers are	e
	printed in decimal
	stored in binary
17 significant figures	39

Even with simple numbers like this, though, there is a catch. We use "base ten" numbers but computers work internally in base two. So fractions that are powers of two (half, guarter, 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. In practice we get sixteen significant figures of accuracy in our floating point numbers. 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. Note for completeness: 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. (If you have no

underlying C compiler that was used to compile the Python interpreter. (If you have no idea what that statement meant, don't worry about it; you don't really need to know this level of detail about Python.) What this does mean is that on most modern PCs you will get at least 17 significant figures of accuracy, but the exact precision may vary. Python does not provide any way for the user to find out the exact range and precision of floating point values on their machine.



This many significant figures isn't so terrible. If you are relying on the seventeenth then you are sunk anyway.

Same basic ope	rations
>>> 5.0 + 2.0	>>> 5.0 * 2.0
7.0	10.0
>>> 5.0 - 2.0	>>> 5.0 / 2.0
3.0	2.5 - Gets it right!
>>> 5.0 % 2.0	>>> 5.0 ** 2.0
1.0	25.0
UCS	41

Let's stick with simple floating point numbers for the time being. It won't take long to get in trouble again. The basic operations behave well enough and use exactly the same symbols as are used for whole numbers.

Note that this time the division of 5.0 by 2.0 gives the right answer, 2.5. There is no truncation to whole numbers.

>>> 4.0 \* 4.0 16.0 >>> 16.0 \* 16.0 256.0 >>> 256.0 \* 256.0 65536.0 \* 65536.0 >>> 65536.0 \* 65536.0 4294967296.0 UCS

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 1.8446744073709552×10<sup>19</sup>.

First note the notation used. Python uses the notation e+19 to mean  $\times 10^{19}$  at the end of a number. This representation is known as "exponential" or "scientific" form. We've been dumped into it because we have reached the limits of accuracy that 17 significant figures can offer.

Second, note that this is not the right answer. There is an error in the value, albeit small relative to the size of the number.

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. So if you are doing mathematics with values that ought to be integers you should stick to the integers, not the floating point numbers.



Now that we're in exponential notation can we continue the squaring further? At first glance, yes we can.



But no. Even in this form, floating point arithmetic has its limits. If we square beyond approximately 10<sup>300</sup> we get an "infinite" answer. Floating point systems have a special code for "number too big to fit" which they casually describe as "infinity". Python prints this out as the three letters "inf".



So floating point numbers, while they can handle fractions (unlike integers) have limits. They are limited in accuracy and range. On the typical PC we get seventeen significant figures and scales between  $10^{-324}$  and  $10^{308}$ .

Progress					
Floating Point numbers	1.25 1.25	5 — 5×10⁵	→ →	1.25 1.25	5 5e5
Limited accuracy Limited range of sizes	(but typically good enough)				
Mathematical operations	a+b a+b	a-b a - b	a×b a*b	a÷b a∕b	a <sup>b</sup> a**b
UCS					47



In this case "wrong" means not precisely correct.



Finally in this review of Python types we will look at text.

Python stores text as "strings of characters", referred to as "strings". ps: See http://www.lipsum.com/ for the history of the "lorem ipsum" typesetting

test text.



Simple text can be represented as that text surrounded by either single quotes or double quotes. Here we use single 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.

The quotes are not part of the text; they simply indicate that the lump of text should be interpreted by Python as a text object.

If we type a string into interactive Python then it responds as usual with that value. Note that Python uses the same single quotes notation to indicate that this is a text object.



Up till now, we have seen no difference between a raw value and a printed value. Integers and floating point number look the same either way. This is because Python doesn't need any syntactic assistance to recognise integers or floating point numbers. It does need help with text, though. A string of characters like "print" might be either the literal string to be evaluated and returned just like a number or a command to be run.

With quotes it is a literal string.

Without quotes it is something that Python will process, such as a command.



The print function outputs the raw text, without any surrounding quotes.



We can also use double quotes around the text. It makes no difference at all to the text object created. Again because of limitations on traditional keyboards we use the same double quote character at the end as the start of the string.

One of the effects of it making no difference is that if we input a string with double quotes Python may well show it with single quotes. This is how Python represents strings. It has no memory of what quotes were used to input it in the first place.



The only condition on using single or double quotes is that you must use the same at either end of the string. You cannot start with one and end with the other.



The flexibility of using either single or double quotes to identify text to the Python interpreter is that we have an easy way to create text objects that have quotes in them. If you want a text object with double quotes in it then define it with single quotes around it. If you want one with single quotes in it define it with double quotes around it.



Python has various facilities for manipulating strings of characters. We will see two at this point. Strings can be joined together with the "+" operator. Note that no spaces are added as strings are joined.



We can also repeat a string by "multiplying it by a number". Note that both "Bang! " \* 3 and 3 \* "Bang! " are valid.

## Progress

Strings

Use quotes to identify (matching single or double)

Use print to output just the value

String operations

UCS

58



Feel free to write your predictions on the notes; it helps stop you cheating with yourself. If you can't understand why you get any of the answers, ask.



So far we have looked at simple, short strings. Suppose we wanted some text that was long enough to require line breaks, or a short piece of text where we wanted to include some line breaks for formatting reasons.

We hit a problem. If we try to create a string the way we have been doing so far the Python system throws an error when we hit the  $[\leftarrow]$  key.



If we can't press  $[\leftarrow]$  to signal "line break goes here" we need some other way to do it. Python uses a common convention (originating in the C programming language) that the pair of characters "\n" represents the "new line character".

The first character is called a "backslash". Note that it is not the same as the forward slash, "/", which Python uses for arithmetic division.

On most modern operating systems line breaks are recorded in the data as an explicit character or set of characters. They don't agree on what the characters should be, but "\n" is what our platforms use.



Note that "\n" is just a way to represent the new line character. There are not two characters there; there's only one.

Internally characters are represented as numbers, and the new line character has a number just like each of the letters.



"New line" is not the only special character like this.

The machines in our public classrooms have had their speakers disabled so you can't heart the beep from "\a" ("**a**larm"). The sequence "\t" gives the **t**ab character. The backslash can also be used to introduce ordinary characters where they would otherwise have special meaning. We can use it to introduce quote marks without worrying about the quotes around the string, for example.

Also, we have to backslash the backslash character if we want it in a string. For interested readers only:

There are more white space characters than new line and tab, by the way. Python supports these less commonly needed sequences too:

\a	bell/ <b>a</b> larm	print('beep\a beep\a')
\b	Backspace	print('abc\bdef')
\e	[Esc]	
\f	Form feed	print('abc\fdef')
\n	New line/Line feed	print('abc\ndef')
\r	Carriage <b>r</b> eturn	print('abc\rdef')
\t	Horizontal <b>t</b> ab	print('abc\tdef')
		<pre>print('ab\tcdef\nabc\tdef\nabcd\tef')</pre>
\v	Vertical tab	print('abc\vdef')

Many of these hark back to the days of teletype printers.

Be careful with [Esc]. It can be used to send instructions to your terminal, rendering it potentially unusable until reset.



But this is fiddly. We want to be able to just hit the  $[\leftarrow]$  key.

Python has some special support for "long strings" where line breaks are likely to be required. If we start a literal string with three single quotes then we can just hit the  $[\leftarrow]$  key the way we would like to. This strings can span as many lines as we want and closes with a matching triplet of quotes.

Note that the string that is created with way is just another string. The triple quotes procedure is just a trick to enter long strings more easily. It doesn't create a new type of string.

What the string is <i>vs.</i> how the string prints		
'Hello,\nworld!'	Hello, world!	
It's not just quotes vs. no quotes!	I	
UCS		65

The new line character emphasizes the difference between the way Python *represents* an object (e.g. a string with its quotes and special characters shown in strange ways) and the way it *prints* that object (which interprets those special characters).



Note that for the long string trick we can use a triplet of either single or double quotes, but they must match at the two ends.



There is no limit to how long a long form literal string can be.

Progress			
Entering arbitrarily long strings Dealing with line breaks	Tripl	e quo '''''	tes
Other "special" characters	\n	\t	
UCS			68

## Exercise

Predict the results of the following instructions. Then check.

- 1. print('Goodbye, world!')
- 2. print('Goodbye,\nworld!')
- 3. print('Goodbye,\tworld!')

UCS



Now we have values we can start comparing them. We can ask if two values are the same, obviously but we can also ask if one is bigger than the other. For numbers this makes obvious sense but for other sorts of values it might make none at all.



For numerical comparisons we can use the symbols provided on the keyboard. If we type a comparison at the interactive Python prompt we are told whether or not the comparison is correct. (We will return to "True" and "False" soon.) Note that we can compare whole numbers and floating point numbers.



Perhaps the most important comparison is to test for whether two values are equal. The operator to do this is a *double* equals sign. The single equals sign is used for something else and we will meet it shortly, but for comparisons two values we use a double equals sign.


Comparing 4 and 5 interactively is hardly useful though, so here's one you may have to think about.

All numerical comparisons		
Python	Mathematics	
x == y	x = y	
X != Y	$x \neq y$	
X < Y	x < y	
$X \leq y$	$X \leq Y$	
x > y	x > y	
$X \ge Y$	$x \ge y$	
UCS	74	

There are six numerical comparisons in total. The "strictly less than" and "strictly greater than" comparisons simply use their symbols on the keyboard ([Shift]+[,] for [<] and [Shift]+[.] for [>] on the keyboards you are most likely to use). The other comparisons use double characters (which must not be split by spaces).



When we compare numbers there is an obvious "right" answer. When we compare strings we use alphabetical order.



But what about mixed case words? Python orders all the upper case letters in front of all the lower case letters.







We have seen interactive Python respond "True" and "False" to our comparison enquiries. These are not just remarks from Python but true values. They are values of a type called "Boolean" which can only take two values: True and False. This new type has the same status in Python as integers, floating point numbers, strings etc. Just as the "plus" operator takes two integers and gives an integer, the "greater than" operator takes two integers and returns a Boolean.



Now that we have booleans as values we can manipulate them. Just as there are operators that combine integers to create integers (1 + 1 gives 2, etc.) there are operators that combine booleans to give booleans.

The first we will meet is "and". This takes two booleans and if both of them are True gives True as a result. If either or both of them is False then it gives False.



Similar to "and" is "or". This returns a True if either or both of the given values is True.



The "or" operator only returns False when *both* its arguments are False.



There is one other boolean operator we need to know about. The "not" operator inverts a boolean value. It turns True into False and *vice versa*.

## >> 1 == 2 False >>> 1 != 2 True

Note that the "not" operator gives us two ways to test for whether two values are unequal.

Progress			
"Booleans"	True	False	
Combination	and	or	
Negation	not		
UCS			85

## <section-header><text><text><list-item><list-item><list-item><list-item><list-item><code-block><table-container><table-container></code>



Before we finish with all this value juggling there is one last thing to address. More complex expressions that involve more than one operator need to have some rules for which operator is dealt with first.

For example there are two possible interpretations for "12+8/4".



Traditionally (or "as human beings") we always interpret this according to the rules on the left hand side of the slide, but for a computer we need to be explicit. We do the division before we do the addition.



Some people say that the division "binds more tightly" than addition. I prefer to say that division goes first.



So, if division goes before addition, then we have an idea of an order that all the operators get executed in. This is called the "order of precedence".



In a nutshell this is it. Exponentiation goes first, followed by remainders, followed by division etc.

Mostly this just does "what you expect".

Parentheses		
Avoid confusion!		
18/3*3	"Check the precedence rules"	
18/(3*3)	"Ah, yes!"	
UCS		92

However, if there is any chance of confusion you should use parentheses (round brackets). Even if *you*'re not confused, if you think it would be easier for your reader to understand your expression with brackets, use them.

## <section-header><text><text><list-item><list-item><list-item><list-item><list-item>Exercise Predict what Python will print when by type the following expressions. Then check 1 12/3\*4 3 3 4 and 1 > 2 or 2 > 1



Now we'll do a more significant example.

This is the start of a build-up to a real Python program. Computer programs run mindnumbingly tedious routines very quickly (so that we don't have to). Unfortunately, to understand just what the computer is going to be doing, we need to understand the mind-numbing bit too. Sorry. It won't last too long.

We are going to get a (poor) approximation to the square root of 2, that is the positive number that when multiplied by itself gives 2. We will use a method called "bisection" and we will do it manually. Later we will learn the Python to automate the process. Bisection works by starting with two estimates for  $\sqrt{2}$ , one too small and one too large.

Each stage of the process starts by calculating the mid-point of the two estimates and seeing if it is too big or too small itself by squaring it and comparing it against 2. If it is too big then we switch our attention to the smaller interval running from the old "too small" estimate to the mid-point which is our new "too large" estimate. If the mid-point is too small then we switch attention to the interval running from the mid-point, which becomes our new "too small" estimate and the original "too large" estimate.

So, each step of the process reduces the size of the interval from "too small" to "too large" by a factor of 2. This converges very quickly but as we are doing it manually we will only do five steps ourselves.

So this slide shows the initial stage. We mark with a red bar the interval between our lower and upper estimates (1.0 and 2.0) and it's corresponding range of squared values (1.0 to 4.0). We start with this interval (that contains  $\sqrt{2}$ ) having length 1.0.



We find the mid-point and calculate its square.



Next we ask if the squared value is greater than 2.0 or less than it. It is greater than 2.0 so we reduce the upper bound to this mid-point. (Otherwise we would have raised the lower bound.)

The interval containing  $\sqrt{2}$  now has length 0.5.



Now we repeat the process.

We find the new mid-point and calculate its square.



We ask if the mid-point squared is greater than  $2 \cdot 0$ . This time it isn't so we raise the lower bound to the mid-point.

The interval containing  $\sqrt{2}$  now has length 0.25.



We do a third iteration. We find the mid-point of this latest interval and calculate its square.



We ask if that square is greater than  $2 \cdot 0$ .

It isn't so again we raise the lower bound to the mid-point.

The interval containing  $\sqrt{2}$  now has length 0.125. The uncertainty over the value of  $\sqrt{2}$  is  $\frac{1}{8}$  of its original size.

Exercise: $\sqrt{2}$ by "bisection"
Three more iterations, please.
UCS 10 minutes 101

We have been using Python as a calculator to determine mid-points, squares and whether numbers were bigger than 2.0. To check out your understanding of python we would like you to do it manually three more times (to get an interval of size 0.015625).



So far we have used Python as a calculator. We needed to do that to get used to some of its properties, but it's capable of so much more.

(Picture © Christian "VisualBeo" Horvat, distributed under the Creative Commons Attribution ShareAlike 3.0 licence. http://commons.wikimedia.org/wiki/File:Calculator\_casio.jpg)



Now we are going to start using it as a real computer programming language. So we need to get a little computer-y.

(Featured computer: a PDP-12

Picture by Bjarni Juliusson, who placed it in the public domain. http://commons.wikimedia.org/wiki/File:PDP-12-Update-Uppsala.jpeg)



We'll get computer-y by looking briefly at how Python stores the values we've been looking at in system memory.

Python stores a value as a record of what type the value is followed by the data corresponding to the specific value. The computer can't interpret that data without knowing what type of value it is representing.



Just for interest, not all programming languages do this. Other require the program to remember what type a lump of system memory contains. Bugs ensue when the programmer gets it wrong and interprets an integer as a floating point number or a string, etc.



Now let's get really computer-y. We are going to start attaching names to our values so we can manipulate them within our programs.

We have seen that if we enter a value at the Python prompt Python responds with that value. If we type in an expression (e.g. 40+2) then Python evaluates it and replies with the expression's value (42 in this case).

Now we will type in a radically different expression. We type in "answer = 42" (n.b. single equals sign and no quotes around the word answer). Python gives no response.

But now we can just type in the word "answer" (without any quotes) and Python evaluates it to have the value 42 that featured in the previous expression.



Let's look at that operation more closely.

1. We start with the name that is going to be attached to a value. Incidentally, if that name was previously attached to a different value then it gets detached from that one and re-attached to this new value.

2. We follow the name with a *single* equals sign. You may recall that when we met the equality comparison operator (the double equals sign) we said we would meed the single equals sign later. This is that moment.

3. Finally we put the value we want the name attached to.

The formal name for this operation is "assignment". The name is assigned the value 42.

Equals signs	
	Comparison: "are these equal?"
_	Assignment: "attach the name on the left to the value on the right"
UCS	108

Just to emphasize: one equals sign  $\rightarrow$  assignment two equals sign  $\rightarrow$  comparison


We typed from left to right. The computer processes the instruction the other way round, though.

1. The expression on the right hand side is evaluated to give the value that will have a name attached to it.

2. Once the value is determined the left hand side is interpreted to get the name to attach. (Later we will meet more complicated left hand sides that require a measure of evaluation themselves.)



In the example we saw the right hand side was a literal value. This is the easiest case where the evaluation is simply "that's the integer 42".



The next level up in complexity is when there is an expression on the right hand side that requires actual evaluation. The expression "44 - 2" is evaluated to a value "integer 42" and after that



But we can go further. Because the right hand side is evaluated *completely* before the left hand side (the name) is looked at, it can contain names itself, including the name that is about to be assigned to!

So, suppose we attach the name "answer" to the value "integer 42". We can then use that name in the right hand side of a following expression.



The process of evaluating the right hand side before the left hand side is rigorously enforced.

1. The expression "answer - 2" is evaluated. The name "answer" appears in it and is evaluated to be its current value, "integer 42". So the right hand side is partially evaluated to be "42 - 2". This evaluation is then completed to give a final value of "integer 40".

2. Then and only then is the left hand side looked at. This contains a name, "answer". That name is currently attached to a different value so it is detached from that and re-attached to its new value. Where this value came from is not relevant.



Let's put named values ("variables") to work.

We'll revisit the square root of two example we met previously. This time, instead of copying and pasting (or retyping) we'll attach names to the values.

We start, as before with initial upper and lower bounds. This time, however, we will attach names to them.

## >>> upper = 2.0 >>> lower = 1.0 >>>

The names we pick are "upper" and "lower". It is always a good idea to pick meaningful names. Avoid the algebraist's approach of calling things "x" and "y".



Next we calculate the mid-point. Again we attach a name to the value and use the two existing names to calculate it.

```
>>> middle = (upper + lower)/2.0
>>> middle
1.5
```

>>>

N.B. The first instruction is all one line.



We need to square the mid-point value and compare it with two to see if it is above (True) or below (False) the square root of two.

## >>> middle\*\*2 > 2.0

True

Because it is above the exact value we reduce the upper bound to the mid-point. Using names for values makes this easy. We simply issue instruction to attach the name "upper" to the mid-point's value which currently has the name "middle" attached to it.

Recall that there is no problem with having more than one name attached to a value. >>> **upper** 

```
2.0
>>> lower
1.0
>>> middle
1.5
>>> upper = middle
>>> upper
1.5
>>> lower
1.0
>>> middle
1.5
```

What matters is that we changed the value upper was attached to rather than lower because of the results of the comparison.



Now it's easy to repeat.

Recall that pressing the up-arrow  $[\uparrow]$  on your keyboard will recall previous lines in Python.

We simply repeat the calculation of middle from the current (updated) values of upper and lower, compare its square to 2.0 and then, depending on whether middle's square is larger or smaller than 2.0 we change the value of upper or lower.

This time middle's square is smaller than 2.0 so we increase the value of lower.



And again.



So we are really caught in a loop. We start with a couple of named values: upper and lower which define the limits of the interval containing  $\sqrt{2}$ .

Then the loop starts.

We calculate the mid-point and attach the name "middle" to it.

Then we square middle and test to see if it is bigger than  $2 \cdot 0$ .

If it is (True) we lower the interval's upper bound by changing the value upper is attached to.

If it isn't (False) then we raise the lower bound by changing the value lower is attached to.

We keep track of our progress by printing the value of the mid-point. We could just as well have printed this as soon as we calculated it, but it will be didactically useful later on to have an explicit instruction here.



Got that?

Let's put it to the test. Can you calculate an approximation to the square root of three  $(\sqrt{3})$  by running three iterations of this loop testing middle\*\*2 against 3.0. Start with lower set to 1.0 and upper set to 3.0.

```
>>> upper = 3.0
>>> lower = 1.0
>>> middle = (upper+lower)/2.0
>>> middle
2.0
>>> middle**2 > 3.0
True
>>> upper = middle
>>> middle = (upper+lower)/2.0
>>> middle
1.5
>>> middle**2 > 3.0
False
>>> lower = middle
>>> ...
```



We've still not delivered on our promise to write a compute program yet. We have variables which make our task easier but we're still not fully automated. We will now inspect the actions we have been taking manually starting with the test we do to see if the mid-point of the interval is too high or too low and what we do as a result of that test.



We square middle and test it for being larger than 2.0 (the number whose root we want).

If that test returns True (i.e. it is larger) then change the upper bound (in variable upper) to have the same value as the mid-point (in variable middle), and otherwise (if it returns False) to change the lower bound (in variable lower) to match the midpoint (variable middle).



In computing we call this the if...then...else... construct.

We run a test (middle\*\*2 > 2.0) and if it returns True then we do something (upper = middle) and otherwise ("else") we do a different something (lower = middle).



So now let's meet our first piece of serious Python syntax.

We take the Python for the test ("middle\*\*2 > 2.0") and precede it with the Python keyword "if". Then we follow it with a colon, ":". The word "if" and the colon indicate that an if...then...else... structure is about to start.

After this comes the set of instructions that are to be obeyed if the test returns True, the "then-block". There is no explicit keyword for "then"; whatever follows the if line is the then-block. All the lines that belong in the then-block are indented. They are set in by a number of spaces (we use four). There can be multiple lines; so long as they are all indented they all belong in the then-block.

At the end of the then-block comes the keyword "else" followed by another colon. This line is not indented, but instead lines up with the if. It does not belong in the then-block, but rather marks the transition from the then-block to the "else-block", the set of lines to be run if the test returns False.

Then comes the else-block itself. This is indented again, and must be indented by the same number of spaces as the then-block. Again, every indented line counts as part of the else-block and the first unindented line (lining up with if and else) marks the end of the whole if...then...else... construct and is obeyed regardless of the test's result.

It's worth noting that the colon at the end of a line is always followed by an indented block. We'll see that pattern again (and again...).



So what does that look like in practice?

The script middle1.py contains a Python script that does the first iteration of our square root bisection system.

We will step through it one block at a time and then run it to see how it behaves.



The script starts with a straight-forward unindented block. These are just lines of Python that get executed. These lines set things up for the test that follows. The first two lines are the initial bounds of the interval containing the square root of two:

```
lower = 1.0
```

```
upper = 2.0
```

The third line is the first of the steps we will eventually repeat, the creation of a midpoint:

```
middle = (lower+upper)/2.0
```

After this block we are ready to move on to the if...then...else... section.



The next line starts with the if keyword, followed by the test, and ending with the colon: if middle\*\*2 > 2.0 :

This starts the if...then...else... construct.

The test we want to ask is whether the mid-point's value is larger than the square root of two?

Because we don't know the square root of two (yet) we set the equivalent test:

Is the square of the mid-point's value greater than two?

That's the Python "middle \*2 > 2.0".



Immediately after the if line comes the then-block. Note that this can include more than one line. In this case we have two lines:

```
print('Moving upper')
upper = middle
```

Each is indented by the same amount: four spaces. Actually, you can use any amount of indentation so long as you are consistent, but four spaces is most common in the Python world and we would encourage you to stick to that. It is *required* that you use the same number of spaces everywhere.



Next comes the line

else:

This is unindented so it marks the end of the then-block and the start of the else-block. Again notice that a line that ends with a colon is followed by an indented block:

print('Moving lower')

lower = middle

The else-block consists of two lines, each indented by the same amount as the thenblock (by four spaces in our examples).



Finally there are some unindented lines at the end of the script. Because they are unindented they do not count as part of the else-block and are run regardless of the result of the text. The if...then...else... construct is over before they start.



We can run this script. It automates for us the first iteration of the process we were doing manually before.



So that was the if...then...else... construct.

Note that what lies between the "if" and the ":" is evaluated to a simple Boolean value (True or False). It can be anything that evaluates like that. It can be a test (the most common case), but it can also be anything else that can be evaluated to a Boolean (including the literal values True and False and any boolean combination of them).



Recall that Python orders strings alphabetically and that the "%" operator returns the remainder, so "3%2" is 1 because 2 divides into 3 with a remainder of 1.



So far we have scripted a single iteration. However, as the name "iteration" suggests we want to iterate it: run it time after time. Our if...then...else... construct sits in the middle of another construct that runs it repeatedly. That's what we want to do next.



Now we could take our script, middle1.py, and run it, edit it to put back the results and run it again. This would be silly.



What we want is some Python syntax that lets us run a block of commands repeatedly. We probably don't want to run for ever, though. Python's way to deal with this is to run some commands while some test returns True.

The command it uses for this is called, naturally enough, "while" and we will use it in a style similar to if....

```
while vs. untilRepeat until...Repeat while...number = 0number != 0upper - lower < target</td>upper - lower >= targetconditionnot condition
```

Be careful. It's very easy to think "loop until". Python thinks in terms of "loop while". Here are some examples of "repeat until…" tests converted into the equivalent "repeat while…" tests. They are essentially opposites. Note that while the opposite of "is equal" (==) is obviously "is not equal" (!=) the opposite of "is less than…" (<) is "is greater than *or equal to…*" (>=). Don't forget the "or equal to" bit. Generally, any Python test can be preceded by the logical negation operator "not".



Let's examine this loop construct in isolation first before returning to our bisection example.

There is a script prepared for you which takes a number, starting at 1, and doubles it repeatedly until it goes over 1,000. We'll take it bit by bit.



We start with the preamble. This has nothing to do with the looping. This is just set-up.



Now we start the looping.

The introductory keyword is "while". This kicks off the whole construct.

The test is that number is less than the limit. Recall that this test must be True for the looping to continue. Because we are increasing number each time the loop repeats (we're doubling it) and leaving limit unchanged eventually this test will return False.

The line ends with a colon, just like "if" and "else".

Recall that while we might think about when the looping should stop ("until..."), Python thinks about when it should keep going ("while...").



The while line is followed by the body of the loop.

This is the section that is going to be repeated while the test continues to return True. Each line of the loop-block is indented by the standard amount (four spaces for us). Note that, again, indentation follows a colon.



After the loop block there is an unindented line. This line will not be run until the looping is finished.



So let's run it.




Don't worry too much if your mental arithmetic isn't up to while4.py.



Now let's return to our bisection example. We are going to put the if...then...else... construct (which narrows our interval) inside a while... construct which will keep repeating that narrowing until we have our answer.



We will take the logic of building our while... loop very slowly this first time. The if...then...else... improves the interval by a factor of two (i.e. it narrows the interval to a half of its previous size). How many of these iterations do we need to do? In other words, what's the test that needs to go after the while keyword?



So, ignoring the Python for a moment, we need to decide how much we want the approximation improved. The quality of the approximation is given by the width of the interval. The width of the interval is simply the upper bound minus the lower bound. In Python the size of the interval is just upper - lower.



Our interval's size is given by upper - lower.

We will make up a target quality. We'll use  $10^{-15}$  for this example. It can be any very small number. Recall that in Python this is written as 1.0e-15.

So the test that the interval is too big (and that we must continue reducing it) is that upper - lower > 1.0e-15

This means that the full while line is

while upper - lower > 1.0e-15 : and this is the line that we will use.



So let's start building our script.

We start with the outer while loop.

We set up the initial values of the lower and upper bounds prior to any refinement.

The next line is the while... line as described before.

Then we start the indented block to be repeated while the approximation is too coarse. This starts with the creation of the mid-point and will continue with the if...then... else... section.

Finally we announce that we are done by printing the eventual mid-point value. This is entirely unindented so is only run after the while... loop is finished.

All we need to do now is insert the if...then...else... block.



And it's actually trivial. We simply insert it. However, it comes one level indented in its entirety by the while... loop. So all we have done is to take the if...then...else... we already have and move it over one more level of indentation.

The if... and else lines are indented once by the while... loop they are in. The thenblock and the else-block are doubly indented (so by eight spaces) because they are indented once for the while... and once for the if...then...else....



And look! It works.



Let's return to the issue of this nested indentation. The best way to think of it is as an analogue of "legalese" where regulations have paragraphs, sub-paragraphs, sub-sub-paragraphs and so on, each of which is more indented that the level before. But its use of indentation is Python's most controversial features. All languages need some mechanism within the language to mark the start and end of these nested blocks of code. C and derived languages use left and right curly brackets ("braces"). Fortran uses expressions like IF and END IF. The shell (the language you type at the Unix prompt) has if statements that end with "fi" ("if" backwards). Its analogue of the while loop uses "do" and "done" to mark the start and end of the loop-block. It would have used "od" but that was already taken by the Unix "octal dump" command. What is interesting is that when programmers work in these languages they typically added multiple levels of indentation to make them easier to read. Python just takes this one step further and makes the indentation syntactically significant.



So let's look closely at the indentation in the script. The while line ends with a colon and is followed by an indented block. The indentation marks the extent of the block. The first line that's not indented is the first line beyond the end of the block.



Within that indented block we have two more lines that end with a colon and introduce blocks indented with respect to those lines (i.e. already indented one level).

Arbitrary nesting	
Not just two levels deep	
As deep as you want	
Any combination	if inside while while inside if if inside if
lice	while inside while
003	100

We have used the example of an if...then...else... construct inside a while... loop with nested indentation. We can do it with any Python construct that uses indentation, nested arbitrarily and arbitrarily deep.

```
e.g. if.. inside if...
number = 20
if number % 2 == 0:
    f number % 3 == 0:
        print('Number divisible by six')
else:
        print('Number divisible by two but not three')
else:
        f number % 3 == 0:
        print('Number divisible by three but not two')
else:
        print('Number indivisible by two or three')
```

For example, we can nest one if...then...else... inside another.

## Progress

colon...indentation

Indented blocks

Nested constructs

Levels of indentation

UCS

158



This is an extended exercise. You may need to take the full fifteen minutes to write it. We are going to implement a script that investigates a bizarre mathematical phenomenon: Take any positive number. Apply the iteration shown in the slide. The "Collatz Conjecture" says that you will always end up looping through the three numbers  $4 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow ...$ 

Starting with 7 you should see this series of numbers:

```
22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1.
```

Once you've got it working, try starting with 47 for a longer list of numbers, going much higher.

Hints:

1. The test to see if a number is even is to see whether or not its remainder is zero when divided by two:

```
number % 2 == 0
```

- 2. Changing number to number/2 is
   number = number/2
- 3. Changing number to 3\*number+1 is

```
number = 3*number + 1
```

You just need to add the while... and if...then...else... syntax.

Comments	
Reading Python syntax	
middle = $(upper + lower)/2.0$	
"What does the code do?" Calculate the mid-point.	
"Why does the code do that?"	
Need to know the square of the mid-point's value to compare it with 2.0 whose root we're after.	

We're now writing real Python scripts. There's one things we can add that will make life a lot easier in the long term: comments.

We can read Python syntax. We can see a line such as

middle = (upper + lower)/2.0

and determine what it is doing. But *why* is it doing it? Why do we want the value of the mid-point?

A comment is a piece of text in the script which does not get executed by Python and which can carry a message explaining the why of the script.



Comments in Python are introduced by the "#" character, which we will pronounce "hash". The comment can be a whole line or part of a line. Everything from the hash to the end of the line is ignored.

```
Commense _ explanation
# Set the initial bounds of the interval. Then
# refine it by a factor of two each iteration by
# looking at the square of the value of the
# interval's mid-point.
# terminate when the interval is 1.0e-15 wide.
USE
```

Comments can be used, as suggested, to give a "why" for a script.

```
Comparing _ page and page
```

They can also be used to enter copyright and licensing statements.



If the script is being edited you can keep a version number or "last edited" date in a comment. Most version control systems can do this for you automatically.

```
Commense _ bagging
# (c) Bob Dowling, 2010
# (c) Bob Dowling, 2010
# feleased under the FSF GPL v3
# fld: root2.py,v 1.2 2010/05/20 10:46:46 rjd4 $
# $Log: root2.py,v $
# Set vision 1.2 2010/05/20 10:46:46 rjd4
# Removed intermediate print lines:
#
# Set the initial bounds of the interval. Then
# refine it by a factor of two each iteration by
# ...
```

If the script is being edited you can also keep a log of changes in a comment. This is not just a "why" but a "how it got to be this way" comment. Again, some version control systems can automatically maintain such a logging comment.



Perhaps the best way to think of comments is this:

If you were given a script written by someone else, what comments would you like to see to make your life easier? Those are the comments that you should write so that you can pass your script to somebody else.

You may think that your script never will be passed on to someone else. However, *you* may be that somebody. Write a script, put is away and don't come back to it for six months. Next time you read it, it might as well have been written by somebody else.

Exercise	
1. Comment your script:	collatz.py
Author	# Bob Dowling
Date	# 2010-05-20
Purpose	# This script # illustrates …
2. Then check it still works!	
UCS	3 minutes 167

Remember the collatz.py script you wrote for an exercise? Add some comments to it.

Comment lines are ignored by the Python interpreter. They should have no effect on the execution of your code. Make sure your script still works afterwards.

Lists (', Jan', ', Feb', ', Mar', ', Apr', ', May', ', Jun', ', Jul', ', Aug', ', ', Sep', ', oct', ', Nov', ', Dec'] (2, 3, 5, 7, 11, 13, 17, 19) (0.0, 1.57079632679489666, 3.1415926535897931)

Now take a deep breath. We are going to introduce a new type of Python object that is one of the most pervasive types in all of Python. Very many Python procedures rely on it.

It's called a "list", a finite sequence of items (often called "elements").



You can usually spot when you ought to be using a list in Python because you get very repetitive scripts that do something to an item, then do the same thing to another item, then to a third, a fourth and so on.

For example, rather than have a ninety-two line script that has a print statement for each of the chemical element names we would be better off with a list of the ninety-two names and an instruction that said "print them".



So let's look at how that's done.

We will start by creating one of these lists, containing the element names. Then we will introduce a Python construct that lets us do something to each element of the list.



So, let's create a literal list. (i.e. one directly typed in.)

A list is a series of items, separated by commas, contained in square brackets. That's how Python represents it when it's output too.

A list is just another Python object so we can assign it a name too if we want.



This is all there is to the representation of a list.

Spaces either side of the square brackets or commas are ignored.



We are going to meet square brackets a lot in the remainder of the course so we will start building up a list of the various things they are used for. First use: they are used to mark the ends of a literal list.



Note that the elements of a list have a specific order. It is the order they are defined with and there is no automatic sorting or reordering based on the values of the items in the list.



Also note that you are perfectly well allowed to have values appear more than once in a list. Repeats are not stripped out.



So what can we do with lists?

Well, we can join them together in a process called "concatenation". Just as we did with strings, we can concatenate them with the "+" sign.



Again, notice that there is no automatic uniqueness. If a concatenation beings two identical values together you end up with a list containing both of them.

```
Empty list

 \sum_{i=1}^{i=1} \sum_{j=1}^{i=1} \sum_{i=1}^{i=1} \sum_{j=1}^{i=1} \sum_{i=1}^{i=1} \sum_{j=1}^{i=1} \sum_{i=1}^{i=1} \sum_{j=1}^{i=1} \sum_{j=1}^{i=1} \sum_{i=1}^{i=1} \sum_{j=1}^{i=1} \sum
```

There's nothing to say that you can't have an empty list. A pair of square brackets with nothing between them (spaces are still ignored) gives an empty list.

Progress Lists	[23, 29, 31, 37, 41]
Shown with square brackets	
Elements separated by commas	
Concatenation	[23, 29]+[ 31, 37, 41]
Empty list	[]
UCS	179

## Exercise

Predict what these will create. Then check.


Now let's start doing some things with our lists. We can ask how long our list is with a new Python function called "len()".



We can also ask for the length of a string.

This counts the characters in the string. Recall that the quotes simply indicate to Python that this is a string; they are not part of the string.



Note that we can't ask for the length of a number. It is a meaningless concept.



len() is our first real Python function, as print is a bit special, so it pays to take a close look.

The function name is followed by round brackets ("parentheses") which contain everything that is going to be fed into the function for it to calculate a result. In this case there is only one argument. It is a list (which contain elements of its own) but the one list is the one argument. Triggering the use of a function like this is called "calling the function".

The function calculates a value from the input(s) it is given in its brackets and when Python interprets the function it uses this value. We say that "the function *returns* a value".





There's two slides of exercises to do in this exercise segment. The first covers lengths of strings...



...and the second the lengths of lists.



We've looked at lists as a whole, but we still need to extract individual items from them.



They key to extracting individual elements is that they each have a position in the list. By declaring the position we can access the item. This position is called the index of the item in a list.

Python counts its indices from zero.

(This is not uncommon in programming languages. The "count from zero" *vs.* "count from one" philosophical battles have been fought long and hard.)

What matters from our perspective is how Python refers to the index. It does it by taking the list (or, more typically, the list's name) and following it with the index in square brackets.

```
Square brackets in Python
(...) Defining literal lists
numbers[/] Indexing into a list
e.g.
>> primes = [2,3,5,7,11,13,17,19]
>> primes[0]
```

See, I told you that we would see square brackets a lot.

The fact that lists use square brackets for literal lists as well as indices is just a coincidence. Later we will see square brackets used for indices on an object created with curly brackets.



Remember that Python counts from zero.

A useful trick of language is to avoid talking about "the first item" or "the second item" and to refer to "item number zero" or "item number one".



What happens if you give an index that runs off the end of a list? You get an error message.



Remember that Python counts from zero. If a list has length four, it has four items in it. These are indexed 0, 1, 2, and 3 so 4 is not a valid index.



But we can use negative numbers!

The index -1 refers to the last element of the list. This can be very useful.



The negative numbers actually work all the way back though this is typically less useful than just refering to the last item with -1.



But you can't go too far back just like you can't go too far forwards.



There is always one zero-or-positive index and one negative index for each entry. An empty list has *no* valid indices.



Square bracketted indices typically follow a list name. What matters is that the thing they follow is a list, whether this is via a name or directly. However putting an index after a literal list is legal but not useful. The author can imagine no use for it at all, but is realistic about the limits of his imagination.

Later we will meet functions that return lists. We can put the square bracketted indices directly after those function calls too.



We use this notation to set the individual values in lists as well as to get them, Just as we can use a simple name on the left hand side of an assignment, we can use an indexed name on the left to set the value of just one element of the named list. Lists are known as "mutable" objects because we can change individual elements within them.

## Progress

Index into a list	['x','y','z']
Square brackets for index	list[index]
Counting from zero	list[ $0$ ] $\longrightarrow$ 'x'
Negative index	list[-1]→'z'
Assignment ucs	list[ 1] = 'A' 200

## Exercise

Predict the output from the following five commands. Then try them.

```
data = ['alpha','beta','gamma','delta']
data[1]
data[-2]
data[2] = 'gimmel'
data
UCS ③ 3 minutes 201
```

Note that two of the commands in this exercise don't produce any output.



Let's return to our challenge. We wanted a list of chemical element names and then we were going to do something with each of the items in the list.

Well, we now know how to create a list of element names:

names = ['hydrogen', 'helium', 'lithium',...]

so now we just need to learn how to do something with each and every one of them. For our specific challenge we want to print it.



We are going to start with a list. Our construct will take the first element of the list and do something with it. If there is a second element the construct will then move on to that item and do the same thing to it. And then to the third, the fourth and so on, until there are no elements of the list left.

We will be reading our way through the list. We will not be removing the items as we go.)



The key is the "do something with that element" phrase.

First, we need some sort of hook to identify the particular element in question. Second we need to identify the block of commands that will be run on the current element. This is Python so the block will be marked by being indented.



So, let's meet the Python construct we will be using: the "for loop".

It gets its name from the first keyword in the line: "for".

This is followed by a name. This should be a new name, not already in use. The name is going to be used to refer to the elements in the list, one at a time, as we will see in a moment. In the case of the slide we are using the name "letter".

The variable's name is followed by a second keyword "in". This is pure syntactic sugar and helps the line read more like English.

After "in" comes the list itself. This can be either a literal list (as shown in the slide) or the name of a list, or (as we will see later) a function whose returned value is a list.

At the end of the line comes a colon, to introduce the indented section.

Next comes the set of commands that are going to be run over each item in the list as an indented block. There can be as many lines as you want; once the indentation is over the block ends.

Note that within this block we can use the variable name created on the first line, "letter" in the case of the slide. The block will be run once for each element of the list and each time it is run the name will be associated with a different item in the list. So, in the case on the slide, the block of code will be run three times. The first time it is run the name "letter" will be associated with the value "a", the second time with "b", and the third and final time it will be associated with "c".



Let's look at this in practice. The script for1.py in your home directories implements the code from the previous slide together with a final line just to prove that the repeated block is finished with.

## Progress

The "for..." loop

Processing each element of a list

for item in items :
 ...item...

UCS

207

Exercise	
Complete the script elements1.py	
Print out the name of every element.	
UCS (V 10 minutes 208	

The file elements1.py contains the Python for a list of all the element names. Complete the script by adding the loop to print out the entries in the list.



Let's go back to extracting elements from lists. We have already seen how to extract a single element from a list by quoting its index in square brackets after the list, or the list's name.

We can also extract parts of lists as lists themselves. These are called "slices" of lists are are created with this variant form of the index.



Let's look at the slice definition. It's two numbers separated by a colon.

The first is the lower index. The element that has this index in the original list becomes the first element in the new list.

The second number is the index of the first element of the original list that is not in the created list. As ever, Python has the approach that the second number is the first index that doesn't get included.



The "off by one" final index system continues to cause some people distress. It does have one useful feature, though. If you concatenate two slices from the same list with matching inner indices then those indices "cancel out" and you get the slice corresponding to the outer indices.



We can omit one of the limits.

If we omit the second, upper limit then the slice is the sub-list starting at the lower index and going to the the end.

If we omit the lower limit then the slice starts at the beginning. Note, again, the the created list stops one short of the index quoted.



This prompts the question: what happens if we omit *both* limits? We get a copy of the whole of the list.





This is really just a variant on the indexing use of square brackets.



We used simple index notation to read an item from a list.

Recall that we use exactly the same notation to refer to "element number two" in the list but this time we place it on the left hand side of an assignment.


This prompts a question. Is there a difference between changing a list one item at a time (which we will call "modifying the list") and simply changing the whole list in one go (which we will call "replacing the list")? There is a difference but it is subtle.



Let's start by looking closely at the "modification" model.

We start by establishing a list, "xyz" with two items in it, the single-character strings 'x' and 'y'.



Now we will modify item number zero in the list.

We will examine the assignment very closely.

The right hand side of the assignment is evaluated first. So Python creates a single character strings 'A' in memory.



Next the left hand side is processed. The list's item number zero is replaced by the freshly minted 'A'. The previous value, 'x', is left behind and Python has internal, automatic mechanisms to delete it from memory if it is no longer refered to anywhere. The posh name for this is "garbage collection". The unused 'x' is "garbage" and the act of identifying it as unused and deleting it to free up Python memory is called "collection".



We do the same thing for item number one in the list, giving it the new value 'B'. We now have the same list object but with both its items changed.



Now let's look at the replacement scenario.

Again, we start by creating a two item list called "xyz". Our starting point is identical.



Now we do the assignment.,

We start, as ever, by evaluating the right hand side. This triggers the creation in Python memory of a whole new list containing 'A' and 'B'.



Then Python processes the left hand side. The name "xyz" is now assigned to this new list and the whole of the old list is unused (and garbage collected).



So we get a different list with the second approach of replacement. Both cases give us a list with the same content, though, so does it really make a difference? Here comes the subtlety I warned you about...



Let's suppose we had two names assigned to the same list.



It we do the modifications one after the other we simply change the content of the list both names point to and so we can change via the name "xyz" and see the results in "abc".



Now we will look at replacement, starting with exactly the same situation.



Assigning "abc" in one go causes the name to point to the new list. But now, instead of the old list being unused, and therefore garbage collected, it is still refered to by the name "xyz".



We have used the simple index notation of the left hand side to modify individual elements of a list. We can also use the slice notation on the left hand side to modify parts of the list. We can even change the length of the list in the process.

```
Progress
Modifying lists values[N] = new_value
Modification ≠ replacement
values[0] = 'alpha'
values[1] = 'beta'
values[2] = 'gamma'
values = ['alpha', 'beta', 'gamma']
```

```
Exercise 1. Predict what these will do.
2. Then run the commands.
>> alpha = [0, 1, 2, 3, 4]
>> beta = alpha
>> gamma = alpha[:]
>> delta = beta[:]
>> beta[0] = 5
>> alpha
>> beta
>> beta
>> beta
>> delta
©
Stinutes 23
```



A very common modification requirement is to be able to add something to the end of a list. In fact, it's very common to create lists by starting with an empty list and building one item at a time.

To do this we have to introduce a new element of Python syntax.

Certain Python objects (for example, lists) have built-in functions, called "methods". We see a simple example of one of these here.

The name "abc" is assigned to a list, initially ['x', 'y'].

We then use this new syntax, "abc.append('z')", to append 'z' to the end of the list.

So what's going on here?



The syntax for a method — a built-in function — is to take

- 1. the object (or more typically a name assigned to the object), followed by
- 2. a dot to act as the glue, followed by
- 3. the name of the method, "append" in this case followed by
- 4. round brackets to contain
- 5. the arguments passed into the function.

Note that we don't seen to pass the list itself in as an argument. Built-in functions know where they came from and have access to the object.



These "methods" are core to the idea of "object oriented programming". While we won't dwell on it too much in this course, there are volumes written on this type of programming.

The UCS offers a course which may be useful to take this aspect of Python programming further:

Object Oriented Programming: Introduction using Python: http://training.csx.cam.ac.uk/ucs/course/ucs-oop



Let's return to the only example we have met so far: the "append()" method for lists. It adds a single element to the list each time it is called.



You cannot use the append() method to add multiple items by putting them in a list. All you get is a "mixed list" that has (in this case) three strings and a list as its four elements.



Mixed lists, while syntactically legal, are almost always the sign of confused thinking. Avoid them. Stick to lists of just one type.

Don't forget that a list of lists of integers is a perfectly sound list. It's a list of a single type: lists of integers. Each of its elements is also a perfectly sound list: a list of integers.



So how do we add a list to the end of a list?

We can use the extend() method which takes a list of elements as its argument and adds them individually to the end.

But there is no need to ever use this!

```
Avoiding extend()
>>> abc = ['x', 'y', 'z']
>>> abc = abc + ['A', 'B', 'C']
>>> abc
['x', 'y', 'z', 'A', 'B', 'C']
UCS
```

Remember that lists can be concatenated with the "+" sign. So we have this much simpler syntax to do it.



There's something worth noticing about both the append() method and the extend() method. Both of them modify the list they are a method of. They don't return a new modified list, they silently modify the list itself.



Let's look at a couple more methods like this.

We start with sort() which takes causes the list it is attached to to become sorted *in place*.

A list can be of any type of item which supports ">", ">=", etc. and it will happily sort.

## Any type of sortable element

243

```
>>> abc = [3, 1, 2]
>>> abc.sort()
>>> abc
[1, 2, 3]
>>> abc = [3.142, 1.0, 2.718]
>>> abc.sort()
>>> abc
[1.0, 2.718, 3.142]
UCS
```

Any sort of list where "<" etc. makes sense can be sorted.



Here's another. The append() method adds an item to the end of its list. How do we add items elsewhere?

The insert() method takes two arguments. The first is the index before which the item is to be inserted and the second is the item itself.



We have met four list "methods" which modify the list itself but don't return any result.

## 1. Predict what this will do. 2. Then run the commands.



Copying items across one at a time with a for... loop is typically overkill. However, if you want to change that number as it gets copied across then it's quite a sensible approach.

This is an example of straightforward copying.



And here's an example of changing the number as it goes across. In this case we square it.

Note that we are using a literal list of the numbers from 0 to 9. There must be a better way to do it than that!



## There is!

Python has built into it a function called range() which generates lists of whole numbers. As ever, it starts at the first argument and ends one short of the second argument. (c.f. slices)



This makes our instructions a little more sensible. More importantly, it makes them more flexible. I can adapt this program to run up to 99 rather than 9 with a simple edit of just one number.



There are other ways to get lists. A method that's often useful when processing lines of data is the split() method on strings. This returns a list of "words" which are the components of the string separated by spaces. It is a very primitive mechanism; there are better but more complex methods elsewhere in Python. For example, it only splits on spaces, not other punctuation.

## Progress Ways to build lists: data[:] slices for loops appending elements range(m,n) function split() string method UCS

252


Here are some hints to help you with the exercise:

1. Run a variable from 0 to 10 inclusive.

To run a variable through a list you will need to use a for... loop.

To get the values being a list of numbers from 0 to 10 inclusive you can either use a literal list [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] or you can use the range (*from*, *to*) function. Recall the strange behaviour about the upper limit of the list produced.

2. Create a list...

In this case, when you are building an "outputs" list from an "inputs" list (0–10), your best bet is to start with an empty outputs list before the for... loop starts and to add an output to it for each run of the for... loop.



I want to take a quick diversion to discuss something that may be coming to mind but which we are *not* going to handle yet.

Image (c) FreeFoto.com: licensed under Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Licence.

Array	's as	s lists	s of I	ists		
0.0	-1.0 -4	4.0 -1.	0.0			
1.0	0.0 -	1.0 0.	0 1.0			
4.0	1.0	0.0 1.	0 4.0			
1.0	0.0 -	1.0 0.	0 1.0			
0.0	-1.0 -4	4.0 -1.	0.0			
[ [0.0,	-1.0,	-4.0,	-1.0,	0.0]	/	
[1.0,	0.0,	-1.0,	0.0,	1.0]	/	
[4.0,	1.0,	0.0,	1.0,	4.0]	/	
[1.0,	0.0,	-1.0,	0.0,	1.0]	/	
[0.0,	-1.0,	-4.0,	-1.0,	0.0]	]	255
003						200

Scientists deal in arrays of data, not just linear lists. Two-, three- and four-dimensional arrays are common.

Plain Python can handle multi-dimensional data, but its facilities are limited. Python would represent a two-dimensional array as a list of lists. The "outer list" would have one row per item. Each item would be a list of the elements in that row.



And don't forget that Python indexes from zero.



Referring to a single row as a "thing" is easy...



...but there is no way to refer to a column with simple syntax.

Numerical Python?					
Hold tight!					
Later in this course, powerful support for:					
numerical arrays matrices					
UCS		259			

But all is not lost!

Later in this course we will refer to a set of Python functions and objects known as "numerical Python" or "numpy" for short. This will solve all our problems. Be patient.



We now return you to your normally scheduled course.

Image (c) Flickr user illustir, released under a Creative Commons licence v2.0. http://www.flickr.com/photos/alper/3257406961/sizes/o/in/photostream/



Let's put lists behind us now and move on to look at something else.

At the moment all our Python scripts have been self-contained. All the data they act on is wired into them. We want to move away from that and have them interact directly with the system. The first example of that will be interacting with files.

So, we want our scripts to be able to read data in from multiple files and write results out to multiple files.



We will start by reading a file. The procedure for this comes in three distinct phases. First we will get our hooks into the file we want to read from. This is the transformation from a name of the file to a Python object that represents the file. This called "opening the file".

Second we will use that newly minted Python object to read the data from the file. Third we will dispose of the Python object corresponding to the file to alert the system that we no longer need access to it. This is called "closing the file".



Let's start with opening the file. This is conceptually the most complicated part of the whole process.

We start with the name of the file. This is just a string. In our case we have the name of a file "data.txt". We need to convert that string into a Python object that will let us access the file. The Python object, internally, will need to know what file it corresponds to and how far into the file we have read. On initial creation, of course, this position in the file (known as the "offset") will point to the very start of the file.

This mapping from file name to the file object itself is handled by a Python function called "open()".



So, what's the Python syntax?

The open ( ) function takes the file name as its argument and returns the Python file object.



How can we use this Python object? How do we read data from the file?



We can read the file's content one line at a time.

The file object, data, has a method readline() which reads one line from the file and returns it as a string.

Note that the "end of line" marker is returned as part of the line (at the end, obviously). Also note that if we call the readline() method a second time we get the *second* line of the file.



What's happening is this: Immediately after creating the file object with the open() function its position pointer points to the very start of the line.



When we call the file object's readline() method we read out the first line (including the new line marker) and the position indicator is changed to point to the start of the second line.



When we call data.readline() again the pointer is moved forwards a second time, this time to the start of the third line.

Each time we call data.readline(), the reading starts at the current position, runs to just after the next end of line character and is left ready for the next lot of reading.



We can read the entire rest of the file in one go with the readlines() method (n.b. the terminal "s"). This returns a list of all the lines. In practice we won't do this as we will meet better methods to do it later.



Once we have read all the data we want from the file (not necessarily all of it) we should close the file to tell the system we no longer need it. This is done with the method close().



I mentioned earlier that Python has a couple of tricks so that you would never need to run the readlines() method directly. This is the first of them.

The most common reason for wanting a list of the lines in a file is so that you can step through them one at a time in a for... loop. Python's trick is that for many type of object where there is an obvious "list view" of that object, you can simply drop the object in to a situation where a list would be expected.

For example, if we drop a file into the list slot in a for... loop, it behaves like this list of lines. The two blocks of Python code behave in exactly the same way.



So let's see a real example. This is a primitive "count the lines" script. All it does is to count the lines in a file.

It starts by setting the counter to zero, as no lines have been read yet.

The first file operation is that it opens the file. This returns a Python file object. The author's habit is to name them after the file name if the file name is embedded in the script but it is quite arbitrary. Instead of "data" it could have been called "input", "file\_whose\_lines\_are\_to\_be\_counted", or "fred".

The second file operation is to read the lines from the file, one at a time. We do this with a for... loop. This is the classic way to read in a text file in Python.

Within the block of the for... loop we act on each line as it comes up. In our case we ignore what's in the line itself, but just increment the counter.

The third file operation is to close the file after we have finished with it.

Finally we print out the number of lines.

## Progress

```
data = open('input.dat')
```

```
data.readline()
```

```
for line in data: ...line...
```

UCS

274



Here's an exercise. It's a serious one that should take some time. (The file treasure.txt contains the entire text of "Treasure Island" by Robert Louis Stevenson and is provided courtesy of Project Gutenberg.) Some hints:

```
1. Open the file.
```

```
file = open(filename)
```

2. Set three counters equal to zero.

n\_lines = 0 etc.

3. Read the file line by line. Recall the Python idiom that if you treat a file like a list it will behave like a list of the lines in that file:

for line in file:

4. Increase n\_lines by one. Please tell me you don't need this hint.

```
n_lines = n_lines + 1
```

5. Increase n\_chars by the length of the line.

Recall that len(string) gives the length of the string.

- 6. Split the line into a list of words. Recall the split() method on strings.
  words = line.split()
- 7. Increase n\_words by the length of the list of words

Recall that len(*list*) gives the length of the list, i.e. the number of items in it.

- 8. Close the file. Recall the close() method on file objects.
- 9. Print the three counters. This will do for this exercise:
- print(n\_lines, n\_words, n\_chars)



Now let's suppose we want to do something with the content of the lines, as opposed to just counting the lines. We immediately hit a problem that reading from files always delivers strings. We can't do arithmetic with strings so we need some way to get from, say, the string 1.0' to the floating point number 1.0.



Python has a set of functions for converting types. Each of these is named after the type it converts into and takes whatever it can as an argument.



Recall that lists are valid Python types. Therefore there is a list() function that attempts to convert its argument into a list.

Strings are converted into lists of characters. File objects are converted into lists of lines.

## Example script sum = 0.0 data = open('numbers.dat') for line in data: sum = sum + float(line) data.close() print sum

Let's see the conversions in practice. Given our data.txt file with one floating point number per line, this script adds up all the values.

It reads each line of the file exactly as it did before, but this time it takes the string of the line and converts it into a float before doing arithmetic with it.



To date we have just been reading from files. Now we want to write to them too. We will open a file again, but this time we will need to declare that we are opening it for writing.



Let's start by looking at the open() function we know already. This takes a file name and opens a file for reading. Actually, it can open a file for reading or writing, and the behaviour is governed by a second argument saying which. If this second argument is omitted then the file is opened for reading. If we want to explicitly include that second argument then the way to declare that the file is to be opened for reading is to set the argument to the letter 'r'.

If we want the file opened for writing then we set that second argument to be the letter 'w'.



The open() function returns a file object as ever, with a pointer set to the beginning of the file. Note that if the file already exists the setting of a write pointer to the start of the file effectively truncates the file to being zero bytes long.



So, how do we write to a file? We start by opening it for writing.



Then we use the write() method in the file object. The read() method takes no argument and returns a line. The write() method takes a line (actually an arbitrary lump of data) and returns nothing.

The current position marker (the offset) is moved to the end of the file.



The write() method does not need to be passed lines.





Just as there is a readlines() method there is a writelines() one too.



Once we have written what we want we must close the file with the close() method. We could get away without closing the file for reading. We *must* close it after writing. There is no guarantee that the data will actually make it to disc until the file is closed.


We repeat: You must *always* close a file after writing.





Rather than bore you with a trivial exercise we'll give a very quick example here. This three line script is a complete "write to a file" Python script.



To do something useful with files we need to read and write data at the same time. The classic example of this is a "filter" which reads in one file and writes out another based on the input's contents a line at a time.

We will write very simple filters right now and look at more complex ones later inth e course when we've learnt a few more tricks.



Here's a straightforward example.

The setup opens the two files. It's a matter of personal choice whether or not you are explicit about the read-only open of the input. The author thinks it helps to contrast the two operations.

Note the explicit close operations. Get into this habit even if, in this particular case, they would have been closed by the script terminating.

Note how ugly the output writing lines are. We can do much better and will see how to later in the course.



Hints:

1. Start with data.txt before trying your script out on the full text of Treasure Island.

2. If the line number is in n\_lines, the line is called line, and the list of words is called words, then the string to output each line is this:

str(n\_lines) + '\t' + str(len(words)) + '\t' + str(len(line)) +
'\n'

3. Work with print() to get the output right and then change to output.write().



Let's look at a common problem in writing a script (in any language). We run a for ... loop using a variable "n". Doing this will overwrite any previous definition of n we had elsewhere in the script. If the script is short then this isn't really a problem. However, as the script gets longer (and they rarely seem to get shorter!) it becomes an increasing risk.



The solution would be to somehow isolate the variable name "n" within the for... loop from any use of the same name outside.



The isolation of the for... loop can't be absolute, obviously. We want to get the limit (11 in this case) in and the results out. But we don't really care what they are called inside the for... loop. We want to pass the value 11 in and get the value of the list out.



We implement this by building our own function. We will pass in the value we want as an argument and read out the value we get as a result. We can then assign this to whatever variable name we want.

(We can also use whatever variable name we want as the input argument too instead of a literal value.)



So let's define a function.

We start with the new Python keyword "def" which starts the definition of a function. This is followed by the name of the function.

Then comes the indicator for the function's arguments. In this introductory course we won't worry about optional arguments and just do functions with a fixed number of arguments. We list the arguments giving them the names that will be used in the definition of the function. These names have nothing to do with any name that may appear outside the function definition. We have our isolation.

The line ends with a colon and what follows, the definition of the function, is indented.



So we follow with our function "body". This indented block carries the actual working of the function. Note that any variable names created within the function (including the one for the argument) are purely internal. If there are variables called limit, answer and n elsewhere in the script they are not touched by this function.



We still have to spit out the function's calculated value. This is done with the new Python keyword "return" which can only be used in a function. This returns the value corresponding to the name answer.

This ends our definition of the function so we cease the indentation.



Now that we have our function defined we still have to use it.

We call these user-defined functions exactly the same way as we use system-defined ones.

Note that the names used outside the function definition have nothing to do with the names used within the definition. It's *values* that are passed in and out, not names.



So we can define our own functions. So what?

There are lots of reasons to use your own functions in your code.

The first reason is clarity. If you extract the nitty-gritty of how do do various operations into functions then you can string the function calls together in the body of your script or program and the whole becomes much easier to read. You can see the wood, because the trees are all packaged up inside functions.

It also lets you write more reliable code. Because your functionality is chopped up into function-sized chunks you can check those pieces individually. If you have one function that reads data from a file, a second that processes the data and a third that writes the processed data out again you can write tests for those three pieces of functionality that won't trip over each other.

Finally, hiving off functionality to functions allows those lumps of functionality to be easily copied into other scripts. (Actually, we don't even need to copy them as we will see very shortly.)



Let's take some real examples, both in the sense that you might really want a function that does this, and in terms of how you might write it. We'll start with creating the sum squares of a list of floating point numbers.

```
Example 1

def norm2(values):
    gum = 0.0

for value in values:
    gum = sum + value**2

return sum
```

This isn't the best implementation in the world, but it is the simplest.

It follows a very common pattern for "accumulating" functions. It sets up an initial value (typically zero for addition, and one for multiplication) and then runs through its input list, accumulating the values from the list. Finally it returns the acculated value to end the function.

Example 1	
print norm2([3.0, 4.0,	5.0])
Ļ	
50.0	
t puthop porm? put	
⇒ python normz.py	
50.0	[3.0, 4.0, 5.0]
169.0	[12.0, 5.0]
UCS	306

This isn't the best implementation in the world, but it is the simplest.

It follows a very common pattern for "accumulating" functions. It sets up an initial value (typically zero for addition, and one for multiplication) and then runs through its input list, accumulating the values from the list. Finally it returns the accumulated value to end the function.

There is an example of this function being used in the file norm2.py. This finds the norm squared of two lists of numbers, once with an explicit list and once with a named list.



Here's another "real world" example. Given a list of values, return the minimum value from the list.



This is an example of a function that won't always work. There is one circumstance when it will fail. What is it?

There is an example of this script in minimum.py. This tries to find the minimum of two lists: once with an explicit list, and once with a named list. There is a third attempt, commented out, which demonstrates how the function can fail.

#### Example 2

print minimum([2.0, 4.0, 1.0, 3.0]) 1.0 \$ python minimum.py 3.0 [4.0, 3.0, 5.0] 5 [12, 5] UCS <sup>309</sup>



This is the generalization of the norm2() function.



Again, this simple Python implementation fails under certain circumstances. The index runs over the length of the first list. What happens if the second list is longer? Or shorter?

There is an example of this script in dot\_product.py. This calculates two dot products, once with literal values, and once with names. It also has two examples commented out that will go wrong in different ways.

#### Example 3

print dot([3.0, 4.0], [1.0, 2.0]))

11.0
\$ python dot\_product.py
11.0
115
UCS

312

```
Example 3 — version 2

def dot(a_vec, b_vec):
    if len(a_vec) != len(b_vec):
        print 'WARNING: lengths differ!'
    Sum = 0.0
    for n in range(0,len(a_vec)):
        sum = sum + a_vec[n]*b_vec[n]
    return sum
```

If there are circumstances under which your function will fail or will give misleading results, it is always a good idea to test your inputs.

Remember: functions get reused. The next user might not be as careful as you, or might not even know the limitation.

Better ways to handle error cases are presented in the "Python: Further Use" course.



This is our fourth and final example. Rather than a simple numerical result, this one returns a list.

```
Example 4

def positive(a_list):
    answer = []
    for a in a_list:
        if a > 0:
            answer.append(a)

return answer
```

Within the function body we use one of our classic means to build a list. We start with an empty list and append() elements to it one at a time.

There is an example script in positive.py. Note that it is quite permissible for an empty list to be returned if there are no positive values in the input.

### Progress

Functions !

Defining them

Using them

UCS

316

# **Exercise** Write a function list\_max() which takes two lists of the same length and returns a third list which contains, item by item the larger item from each list. list\_max([1,5,7], [2,3,6]) $\longrightarrow$ [2,5,7] Hint: There is a built-in function max(*x*, *y*) which gives the maximum of two values.

If you want some hints for how to solve this exercise, look at the third and fourth worked examples again.

Hints:

- The third example demonstrates how you use the index to move through two lists in parallel.
- The fourth example demonstrates starting with an empty "answer" list and growing it an item at a time for each round of a for... loop.
- Python has a function which returns the maximum of two simple values.

```
>>> max(1,2)
```

```
2
```

```
>>> max(4.0,-5.0)
```

4.0

You cannot use it directly on a list.



To date our functions have all returned a single value, even where that value was a list. For example we might have a function that returns the minimum value from a list and a second function that returns the maximum. Why can't we have a function that returns *both* at the same time?

A list of two elements is not an appropriate type to return. The pair of values is just that: a pair of values. There's no reason why they should come in a particular order. There's no concept of the third item in the list.



We should have no problem thinking about the body of the function by now. But what do we do with the return statement to return two values at the same time?

We do it by returning a pair of values. Python indicates these by separating them with a comma. This pair is typically surrounded by brackets for clarity, but actually it's the comma that's the active ingredient.

There is an example of this in the script minmax.py.



So we can emit a pair of values from the innards of the function. How do we pick up those values on the outside when we use the function? We use exactly the same commas and brackets notation as we did before. There is an example of this in the script min\_max.py.



There's a posh name for these comma separated collections of values: "**tuples**". The word comes from the name given once we get past "triples" for three items together: "quad**ruples**", "quin**tuples**", "hex**tuples**", etc. We meet them often enough for tem to deserve a few slides of consideration.

Tuples ≠ Lists		
Lists	Tuples	
Concept of "next entry"	All items at once	
Same types	Different types	
Mutable	Immutable	
UCS		322

Tuples are not quite the same as lists. In fact, they differ very significantly from lists in a few technical ways, but the most important difference is conceptual.

A list is used for a sequence of numbers where there is some concept of successor; each item naturally follows the one before it. A natural question to ask of a list is "is there a meaningful way to extend the list?"

A tuple is used when all the items happen "at once".

In a list, where there is a concept of a sequence, the items tend to be all of the same type. In fact we recommend that you only use lists with all items of the same type. In a list where there are just a number of items grouped together, there is no such obligation.

Finally, there is an important technical difference. We saw with lists that we could change individual elements and had an entire section contrasting modifying a list with replacing a list. Tuples are immutable.

Tuple examples	
Pair of measurements of a tree	
(height,width)	(7.2, 0.5)
(width,height)	(0.5, 7.2)
Details about a person	
(name, age, height)	('Bob', 45, 1.91)
(age, height, name)	(45, 1.91, 'Bob')
UCS	323

Here are some examples of natural use of tuples.

Suppose we are measuring trees. We measure their width and heights. These two number are related (same tree) so we pair them up as we sling them round the program, but they could come in either order. There is no natural order for these two numbers.

We could handle three pieces of data about people in our program, for example their names, ages and heights. These are three different types of data and can come in any order.

## Progress

Tuples

"not lists"

Multiple values bound together

Functions returning multiple values

UCS

324


Copy the min\_max function from min\_max.py.

The exercise is to add an arithmetic mean to the values returned (turning a pair into a triple). To calculate a mean:

1. Set up a sum variable before the for... loop with initial value 0.0.

2. Within the loop add each value encountered to sum.

3. After the for... loop is complete (so not indented) calculate mean as sum divided by the length of the list of values.

4. Return the three values rather than just two.



We'll take a quick break from functions fro a moment. Now that we have tuples we ought to look at what else we can do with them.

Suppose we want to take some values (e.g. from a tuple) and substitute them into a string, mail-merge-style.



Let's start with a single substitution.

We take a string containing the magic code "%s" marking where we want the string to be inserted. It will substitute for the %s.

We follow the string with the substitution operator, "%". This has nothing to do with the arithmetic use of the same character.

We follow the substitution operator with the string to be inserted. The "%s" means that the substitution is expecting a string and a string must be provided.

The result is the original string with "%s" replaced.



We can do exactly the same thing with a "%d" to indicate an integer.



And this is where tuples come in. Suppose we want to substitute a string and an integer. If we follow the substitution operator with a tuple then the markers in the string get replaced in order from the tuple.



In practice we might see something like this.

Our data comes as a list of tuples (or something treated like a list). We can use a tuple of variable names to identify these values in a for... loop.



Trouble is, t produces really ugly output. Typically with lists of data like that we want them aligned in columns. Numbers, typically, get right aligned for easy comparison too.



We have a solution. The substitution operators have a set of modifiers that let us change the details of the substitution. For example,

The simplest are for the strings. Adding a number between the "%" and the "s" specifies how many characters should be assigned to the string. The string is right aligned. If we specify a negative number it is left aligned. (These defaults make more sense for numbers). If the string being inserted is too long then it just overflows; it does not truncate.



There is similar formatting for integers. There is an additional option for integers where a "0" is inserted before the width specifier. This pads the number out with leading zeroes.



We now have everything we need for formatted output.



Finally, we need to look at floating point numbers. These have many more options and we will restrict ourselves to just the most useful here. Note that truncating a floating point number causes it to be rounded.



We have taken a quick tour of formatting and string substitution using tuples (or a single for the simplest cases). There is a fuller set of formatting codes as a separate hand out.

Exercise					
Complete the script format1.py to generate this output:					
Alfred Bess Craig Diana ↑ 1	46 24 9 100 ↑ 9	1.90 1.75 1.50 1.66 ↑ 15			
UCS			S minutes 33	7	

Edit the script format1.py to complete this exercise. I suggest you attack the problem in stages.

- 1. Get the basic %X symbols right.
- 2. Then work on the name column and get it right
- 3. Then work on the age column
- 4. Add some spaces to get the height column right.
- 5. Format the height column for two decimal places.

Reusing our functions				
Want to use the same function in many scripts				
Copy?	Have to copy any changes.			
Single instance?	Have to <i>import</i> the set of functions.			
UCS		338		

Now let's get back to the idea of reusing a function. We can reuse a function within a script easily. Its definition is written once near the top of the script and we use it multiple times within the script. If we change (e.g. fix) the function definition, all the points in the script that use the function immediately benefit.

Now suppose we had written a really useful function that we wanted to use in lots of different scripts.

We can, of course, just copy the function's definition from one script to another. However, if we change (fix) the function definition in one script we have to repeat the edit in all our scripts.

What we want is a mechanism to use a single definition in multiple scripts. This is called "importing the function".



Let's do this as a worked example.

In an earlier exercise we wrote a function that generates (simultaneously) the minimum and maximum of a list and returns it as a pair. We have an example of this in a script called five.py. We are going to split the definition of this function out from the script that uses it.

### \$ python five.py

(1, 5)



The first thing we do is to cut and paste the definition into a different, new file. We will call it utils.py (short for "**util**ities").

The script five.py will no longer work. It cannot find the definition of the min\_max() function it uses.

#### python five.py

```
Traceback (most recent call last):
   File "five.py", line 3, in <module>
      (x, y) = min_max(vals)
NameError: name 'min_max' is not defined
```



So now we modify five.py to import the min\_max() function from utils.py. First, we tell the script where to get some more functions from. We do this with the command

import utils

This causes Python to go looking for a file called utils.py which contains functions. Don't worry about where it goes looking; it includes a set of system locations and your current directory.

On its own this isn't sufficient.

#### \$ python five.py

```
Traceback (most recent call last):
```

File "five.py", line 4, in <module>

```
(x, y) = min_max(vals)
```

NameError: name 'min\_max' is not defined

We need to tell Python that min\_max() is supposed to come from that import. (There may be several imports or it may be meant to come from the current script, or even the system.)



We indicate that min\_max() now comes from the utils.py file by prefixing "utils." to its name.

Now it works again:
\$ python five.py
(1, 5)



This collection of functions in a .py file is called a "module". Actually, a module can contain more than just functions but that's what we are going to be most interested in for this course.

A module is a collection of functions, types of object and various parameters, all bound together, brought in my a single import statement and all with the same dotted prefix to identify where they came from.

System modules					
OS	operating system access				
subprocess	support for child processes				
sys	general system functions				
math	standard mathematical functions				
numpy	numerical arrays and more				
scipy	maths, science, engineering				
CSV	read/write comma separated values				
re ucs	regular expressions				

A huge number of modules exist built in to Python, or typically provided alongside it. Here are just a few of the more useful ones.

Python keeps its language simple by hiving off most of the complexities of special circumstances into modules that you only import if you need that particular piece of functionality.

"There's a module for that" is the standard answer to almost all "how do  $\mathsf{I}\ldots$ " questions in Python.



Let's take an example. We will work interactively here just for convenience. Python itself does not support most mathematical functions. These are in the "math" module (beware American spelling). So if we want the square root of a real number we need the math.sqrt() function, i.e. the sqrt() function from the math module.



There are a couple of short cuts that we want to advise you away from. These are syntactically legal but lead to confusion and the author of Python, Guido van Rossum, allegedly regrets ever having permitted them.

You can import a single function from a module and then use it without identifying which module it comes from.

Don't do that.



You can even import *all* the functions from a module and use them without identifying the module they come from. Really don't do that.

```
Do do this
>>> import math
>>> help(math)
Help on module math:
NAME
math
DESCRIPTION
This module is always available. It
provides access to the mathematical
functions defined by the C standard.
```

So how do you find your way around a new module?

One of the things that should be built in to a module is its own documentation. You may request help on any imported module by issuing the Python command help() on the module name. The module must be imported before you ask for help on it.

# Progress

"Modules"

System modules

Personal modules

import *module* 

module.function(...)

UCS

349

# Exercise

- 1. Edit your utils.py file.
- 2. Write a function print\_list() that prints all the elements of a list, one per line.
- 3. Edit the elements2.py script to use this new function.

UCS	✓ 5 minutes	350
-----	-------------	-----



So now we can start looking at the modules that come with every Python implementation.

The sys module provides the hooks for interacting with the system in an operating system neutral fashion. (There is a separate module for the operations that do depend on the operating system.)



So, what's in sys?

First, we will look at two objects (rather than functions) that are very useful if you write in the classic "filter" style:

python script.py < input\_file > output\_file

The object sys.stdin corresponds to the "standard input" (input\_file in our example) as an already opened file object. The sys.stdout object is the equivalent for the "standard output" (output\_file in our example).



For example, here is a complete Python script for copying one file to another line by line.



Note the usual trick with an open file object: if we treat it like a list it behaves like a list of lines. The sys.stdin object is just an open file. The only difference is that it was opened for us.



Similarly, we treat sys.stdout as an open file (opened for writing). We don't need to open it; the system has done that for us.



We can now copy a file. Great.



Now copying a file is pretty much pointless. We have cp for that.

However, the general shape of the script opens up the route to two different, very commonly needed operations.

The first is where we change the lines, or process them in some way.

The second is where we only write them out again if some criterion is satisfied.

An extreme third case is where we gather statistics as we go and print them out only at the end.



The standard script for modifying a line would be this. Notice that we tend to separate the line-by-line rewrite and the process of running through the lines by splitting the rewrite off to a function.



Similarly, this is the model for optionally writing out the line or not.

Progress			
sys module			
sys.stdin sys.stdout	Standard input Standard output		
"Filter" scripts	process line-by-line only output on certain input lines		
UCS	360		
Exercise			
--	--	--	--
Write a script that reads from standard input.			
If should generate two lines of output:			
Number of lines: MMM Number of blank lines: NNN			
<pre>Hint: len(line.split()) == 0 for blank lines.</pre>			
UCS S minutes 361			

Blank lines may have spaces on them. The best test for blank lines is to take the line and to split it into "words". If there are none, count the line as blank.



Now let's look at another facility that the sys module gives us.

To date we are setting our parameters explicitly in the script itself. We really want to enter them on the command line.



The sys module provides an object sys. argv which is a list of all the command line arguments. We can see this with a trivial script that prints it out.

We should notice a couple of significant points:

The name of the script itself is item zero in the sys.argv list.

All the items on the command line are presented as strings.



Because all command line arguments are presented as strings we can't treat numerical arguments as numbers straight away.



We have to convert them to the correct type. We have already met the type conversion functions. If we want a floating point number on the command line we use the float() function to convert from the given string to the desired float.



Manual parsing of the command line will do for simple scripts. There is a module dedicated to parsing the command line called "argparse". This is more suitable for slightly more experienced scripters, but is exceptionally powerful.



Here's a general approach for scripts that process the command line. The important bit is that the parsing of the command line from string to directly usable values should be split off into a function, which can then be independently tested.



So, let's go. We will write a "proper" program that reads input from the command line and uses it to control its output.

We will consider as our goal a script that takes a floating point number, r, and an integer, N, from the command line and supplies N points (x,y) uniformly distributed along the curve  $y=x^r$  on standard output for x ranging from 0.0 to 1.0 inclusive.



Unsurprisingly, we are going to split it up into functions. Splitting up a problem into components and implementing each component as a function is the key to successful programming. There are two components to our problem. We need to get the command line arguments into forms we can use: one float and one integer. We also need to take these two values and output the corresponding points.





The first function has to parse the command line. We are expecting two arguments so we simply convert them and return a pair (tuple) of the two values. If there are not enough command line arguments or if they cannot be interpreted as the right sort of number then this function will fail and the script will halt.



We write a simple test. The parsing of the command line has to return objects of the correct type and value. So we simply print out their values from within a substitution, which will fail if they are not of the expected types.



It works!

2a. Write a function that takes (r, N) as (float, integer) and does the work.



The second function takes a float and an integer (presumed already converted from the argument strings) and outputs the data we want. Note:

• range(0, num\_points) gives num\_points as desired, but its maximum value is num\_points-1. Because of this we divide by num\_points-1 in the following line and not num\_points.

• index starts as an integer, as does num\_points-1. We explicitly convert both to floats prior to dividing one by the other to make sure we get a float afterwards. (If we did it in integers every value except the last would be 0.)

Our function does not need to return any value because it is just printing output and doesn't need to report back.



Next we need to test our function.

We run it with values passed explicitly in the script. Its specification is that it must produce a certain number of points satisfying a power law. So we have two checks we need to make. Does it produce the correct number of points and are they mathematically correct?



Yes, they are. It works.

```
3. Combine the two functions.
import sys
def parse_args():
    pow = float(sys.argv[1])
    num = int(sys.argv[2])
    return (pow, num)
def power_curve(pow, num_points):
    for index in range(0, num_points):
        x = float(index)/float(num_points-1)
        y = x**pow
        print '%f %f' % (x, y)
(power, number) = parse_args()
power_curve(power, number)
```

Now we trust our two functions we combine them to create the script's final functionality:

- (1) parse the command line to get the power and number of points
- (2) print that many points on the power curve

## Progress

Parsing the command line

sys.argv

Convert from strings to useful types

int() float()

UCS

377

Exercis	е
Write a script t numbers and p	hat takes a command line of prints their minimum and maximum.
Hint: You hav Reuse it	re already written a min_max function.
UCS	S minutes 378



We have seen that the system modules come with their own help. What does ours come with?

We can ask for help and we get a minimal, automatically generated help text. We want to be able to add to this.



We can also ask for help on specific functions in our module and we get just the basic information there.

We want to be able to add help text to individual functions as well as the module as a whole.



We will start by annotating an individual function.



What we will do is simply place a string immediately after the def line and before any of the active lines in the function's definition. (Comments don't count.) Because this is often a long string it is traditional to use triple quotes. It doesn't matter; it's just a string.



Now if we ask for help on that function we get the text we inserted.



How do we annotate the module as a whole?

We add another string to the file, this time before *any* of the active lines.



And we get the text out again when we ask for help on the module.

Progress	
Annotations of functions of modules	
"Doc strings"	
help()	
UCS	

UCS 3 minutes 387	
Annotate your ut ils.pv and the functions in it.	
Exercise	



We now have just about enough Python to write some serious scripts. We need one more feature, and we will meet it by looking at how to do data processing. First of all we ought to look at the sorts of files that contain our data. What format is the data in?



A very common, and very useful format is called "comma separated values". This is usually marked by a suffix ".csv" on the file name. It is a common interchange format for spreadhseets.

Each record is a row. Each column is separated from its neighbours by a comma. Sometimes the records are in quotes.



Here's a quick way to chop up a line at the commas. The split() method takes an optional argument which is the character to split on. Note that the strings in the list have some strange spaces in them. Don't worry; the float() conversion function can handle them.



This is a straightforward conversion.



Don't push the simple split() trick too far, though. There are many cases it can't cope with. If you want to handle CSV files for real you should use the csv module written for just that purpose.

The csv module would work like this. Don't worry about the specifics; there's a proper coverage of the module in the "further topics" Python course.



So how we can read tabular or columnar data how can we store it within the program? Let's consider a case where we want to map from some text key or ID to a tuple of data.



Let's start with a simple case where we map from an id string to a single string value as opposed to a tuple.

```
Not the same as a list...indexname0Joe1Fred2Bess...'...'.........
```

This mapping from string id to value is different from a list. A list is indexed by positions. We need to index by string.
but similar: a "dictionary"				
id	name			
A101 G042 H003	Joe Fred Bess	names['G042'] = 'Fred'		
{'A101':'Joe', 'G042':'Fred', 'H003':'Bess',}				
UCS		397		

We are going to use a different structure. We want something that takes an arbitrary Python object (rather than just an integer) and looks up a corresponding value. Python has such a type, called a "dictionary".



A dictionary maps from an arbitrary Python type (strictly speaking, any *immutable* Python type) to an arbitrary (mutable or immutable) type.

The jargon is that instead of an index a dictionary has a "key" which it maps to a "value".

We can map from strings to strings (a very common case), or from strings to tuples (which we want to do here).



So how do we build a dictionary?

We can create it all in one go as shown. The dictionary is delimited with curly brackets (as opposed to a list's square brackets) and the individual elements are separated by commas, just like a list. The elements themselves, however, are composite. Each is the key/value pair separated by a colon. In a list the order they are specified defines the index. With a dictionary, where we use the key rather than an index, we have to quote both parts.



The alternative approach is to create an empty dictionary with just the pair of curly brackets and then to add the elements one at a time.



So here's an example of a dictionary being created. Note that because there is no numerical indexing there is no natural ordering either. Just because I enter the key:value combinations in one order doesn't mean it stores them in that order.

```
Example — 2
>>> data['A101']
'Joe'
>>> data['A101'] = 'James'
>>> data
{'F042': 'Fred', 'A101': 'James'}
usuble conductors of the second second
```

We get values out of a dictionary by quoting the corresponding key. They key appears in square brackets, just as we did with a list. We can change the value corresponding to a key just like we did with a list.



Note that while we use curly brackets to define a literal dictionary, we still use square brackets to resolve a key to a value in it.

(The only reason Python uses curly brackets rather than square brackets for literal dictionaries is that otherwise the Python interpreter would not be able to distinguish "{}" for an empty dictionary and "[]" for an empty list.)

```
Example — 3
>>> data['X123'] = 'Bob'
>>> data['X123']
'Bob'
>>> data
{'F042': 'Fred', 'X123': 'Bob',
'A101': 'James'}
ucs
```

We can add additional items into a dictionary using the same syntax as we used to change them. Because there is no concept of order, there is no concept of "appending"; we are just adding additional values.



## Exercise

Write a script that:

- 1. Creates an empty dictionary, "elements".
- 2. Adds an entry 'H'  $\rightarrow$  'Hydrogen'.
- 3. Adds an entry 'He'  $\rightarrow$  'Helium'.
- 4. Adds an entry 'Li'  $\rightarrow$  'Lithium'.
- 5. Prints out the value for key 'He'.
- 6. Tries to print out the value for key 'Be'.

UCS

3 10 minutes 406



Let's move forward from that example to look at populating dictionaries from files. You have a file in your home directories called elements.txt. This contains 92 rows of data in 2 columns: the symbol and name for each chemical element. We want to create a dictionary called symbol\_to\_name that contains equivalent data.



Let's see how we would do it.

We'll start with an open file to read the data from and an empty dictionary to write the data to.

Then we run through the data in the file a line at a time, using the string split() method to carve the line up into its two components.

For each line we take those two components as the key and value and ad them to the dictionary.

Finally, after running through the file, we close our input file.



We can do exactly the same to read in our keys  $\rightarrow$  names table too.

```
Worked example -- 4
data = open('names.txt')
key_to_name = {}
for line in data:
    [key, person] = line.split()
    key_to_name[key] = person
data.close()
ucs
```

410

The code is equivalent with just some names changed.



411

This is an obvious candidate to be made a function.



We know what our input should be: the file name.



So we can write the def line and modify the script to use the input variable..



We know what the output should be: the dictionary.



So we can give that a nice generic name  $(x_to_y)$  and change the internal variables to match (x and y).



And we add the return line to hand back the dictionary.



Here's a "skeleton" of the script you need to write:

```
# Read in the master dictionary
import utils
symbol_to_name = utils.filename_to_dict('elements.txt')
# Open the input file
input = ...
# Run through the data file one line at a time
for line in input:
    symbol = line.strip()
    # Look up the name of the element with this symbol
    name = ...
print name
# Close the input file
input....
```

All you have to do is to fill in the three blanks.



How can we tell what keys are in a dictionary?

Specifically, what happens when we want to run through all the keys in the dictionary? We use the Python magic of "treat it like a list and it behaves like a list". In the case of dictionaries it behaves like a list of the keys.

"Treat it like a list"				
"Treat it like a list and it behaves like a (useful) list."				
File	$\rightarrow$	List of lines		
String	$\rightarrow$	List of letters		
Dictionary	$\rightarrow$	List of keys		
UCS			419	

We've seen this before. Files behave like this list of lines and strings behave like this list of letters. Dictionaries behave like the list of keys.



So we can use a dictionary in a for... loop by looping through all its keys and then looking up the corresponding value in the body of the loop.



What happens if we ask for a key the dictionary doesn't have? Obviously we get an error. We get an error very similar to the error we get if we ask for an out of range index in a list. Instead of being an "IndexError", a dictionary returns a "KeyError".



So we need to be able to tell in advance wither a key is in a dictionary. We do this using the "treat like a list; behave like a list" magic. We can ask if key in dictionary just like we can ask if item in list



We can make the change to a list literally, of course, with the type converter function  ${\tt list()}.$ 





Write a function that takes a dictionary as its argument and returns the "reversed" dictionary as its result.

To do this write the def line to take a dictionary  $x_to_y$ .

Start your function body by creating an empty dictionary  $y_to_x$ .

Run through the keys of  $x_to_y$ , calling the key x.

For each x look up the corresponding y in the given dictionary  $x_to_y$ .

For that x and y, add an entry to the  $y_to_x$  dictionary.

Once the loop is complete return the y\_to\_x dictionary.



Let's finish with one last, serious example.

We are going to analyze some text and count up how often each word in the text appears.

## Word counting algorithm

Run through file line-by-line Run through line word-by-word Clean up word Is word in dictionary? If not: add word as key with value 0 Increment the counter for that word Output words alphabetically

427

This is what we are going to do.

We will require the file to be counted to be given on the command line.



We need to import the sys module to get at the command line arguments in sys.argv.



Next we run through the data pulling apart the words. This is a very crude analysis so we suppose a simple function exists that cleans up a word: stripping any punctuation that might have come along for the ride, converting everything to lower case, etc. We will need to write this function.



Here's a simple cleanup function. All it does is convert the word to lower case. If you want a better function that strips out punctuation, go to the course opn "regular expressions".



Now we change the dictionary. If this is the first time we have ever seen the word, we have to add an entry to the dictionary. Because we will be incrementing the dictionary value in a moment we set it to zero on creation.



As soon as we have finished with our nested loops running through the words, we close the data file.

Our specification wanted us to run through the words in alphabetical order. The order we get them from a dictionary by treating it like a list is essentially random, so we create a list and then sort it. This is the alphabetically ordered list of all the words that appear once or more in the file. Each word only appears once in the list; the frequency with which they appear in the data file is in the dictionary *value*.


Then we print out all the words and values.



There is a script prepared for you with this worked example in it. Run it, look at the output. Discuss what changes you would make to im

Run it, look at the output. Discuss what changes you would make to improve the script.

And we're done!	
Python types	
Python control structures	
Python functions	
Python modules	
ucs t	and now you are ready to do things with Python <sub>435</sub>

And that's it!



Unless you want more, of course.