An introduction to Python
for absolute beginners

Bob Dowling
University Information Services
scientific-computing@ucs.cam.ac.uk

http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/PythonAB

1

Welcome to the Computing Service's course "Introduction to Python".

This course is designed for people with absolutely no experience of programming. If you have any experience in programming other languages you are going to find this course extremely boring and you would be better off attending our course "Python for Programmers" where we teach you how to convert what you know from other programming languages to Python.

This course is based around Python version 3. Python has recently undergone a change from Python 2 to Python 3 and there are some incompatibilities between the two versions. The older versions of this course were based around Python 2 but this course is built on **Python 3**.

Python is named after Monty Python and its famous flying circus, not the snake. It is a trademark of the Python Software Foundation.
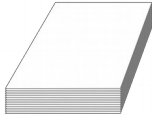
But first…

 Fire escapes

 Toilets

 Notes

 No drinks or snacks in the room, please.

 Signing in

 Feedback

2

Course outline — 1

Who uses Python & what for
What sort of language it is

How to launch Python
Python scripts

Text
Names for values
Reading in user data
Numbers
Conversions
Comparisons
Truth & Falsehood

3

Course outline — 2

Assignment
Names

Our first "real" program

Loops
if… else…

Indentation

Comments

Course outline — 3

Lists
Indices
Lengths
Changing items
Extending lists

Methods

Creating lists
Testing lists
Removing from lists

for… loop

Iterables
Slices

5

Course outline — 4

Files
Reading & writing

Writing our own functions

Tuples

Modules
System modules
External modules

Dictionaries

Formatted text

6

Who uses Python?

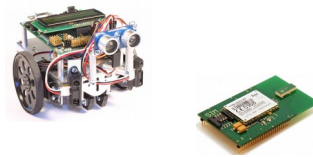On-line games

Web services

Applications

Science

Instrument control

Embedded systems

`en.wikipedia.org/wiki/List_of_Python_software`

So who uses Python and what for?

Python is used for everything! For example:

"massively multiplayer online role-playing games" like Eve Online, science fiction's answer to World of Warcraft,
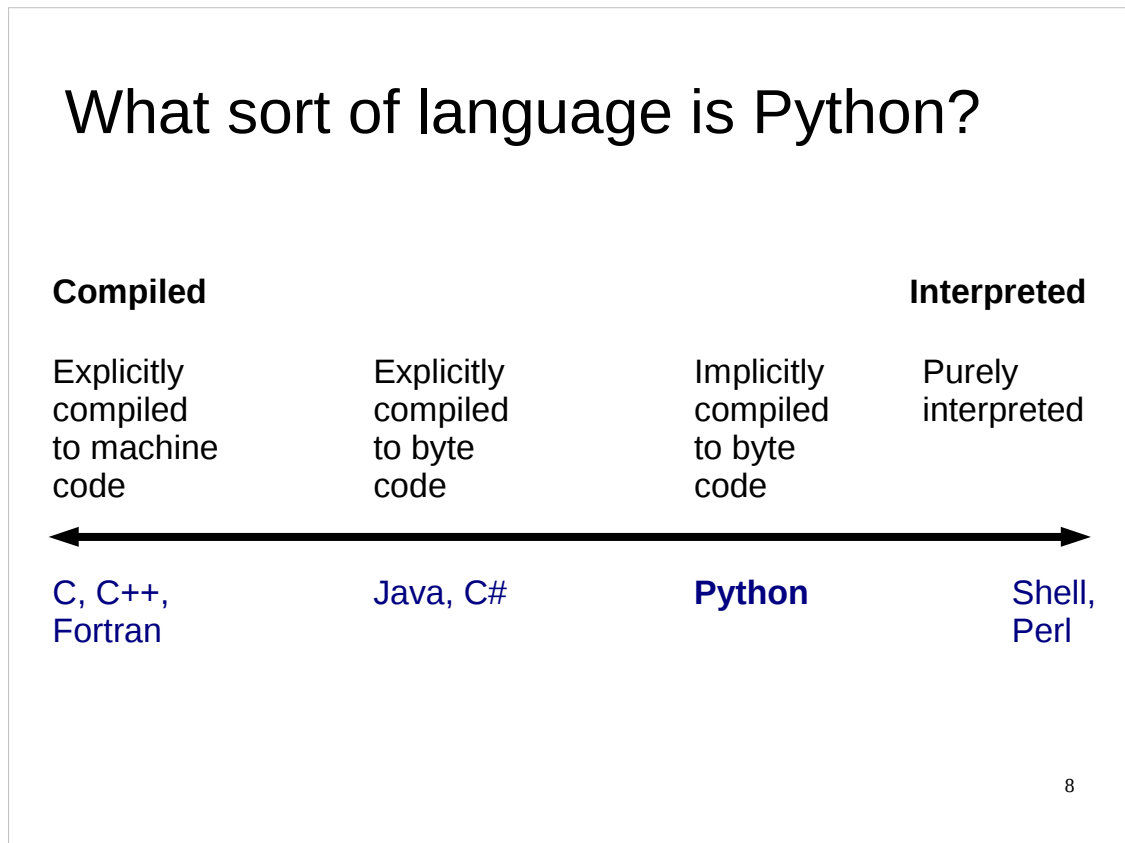
web applications written in a framework built on Python called "Django",

desktop applications like Blender, the 3-d animation suite which makes considerable use of Python scripts,

the Scientific Python libraries ("SciPy"),

instrument control and

embedded systems.

What sort of language is Python?

Compiled			Interpreted
Explicitly compiled to machine code	Explicitly compiled to byte code	Implicitly compiled to byte code	Purely interpreted

$$\longleftrightarrow$$

| C, C++, Fortran | Java, C# | **Python** | Shell, Perl |

8

What sort of language is Python? The naïve view of computer languages is that they come as either compiled languages or interpreted languages.

At the strictly compiled end languages like C, C++ or Fortran are "compiled" (converted) into raw machine code for your computer. You point your CPU at that code and it runs.

Slightly separate from the strictly compiled languages are languages like Java and C# (or anything running in the .net framework). You do need to explicitly compile these programming languages but they are compiled to machine code for a fake CPU which is then emulated on whichever system you run on.

Then there is Python. Python does not have to be explicitly compiled but behind the scenes there is a system that compiles Python into an intermediate code which is stashed away to make things faster in future.

But it does this without you having to do anything explicit yourself. So from the point of view of how you use it you can treat it as a purely interpreted language like the shell or Perl.
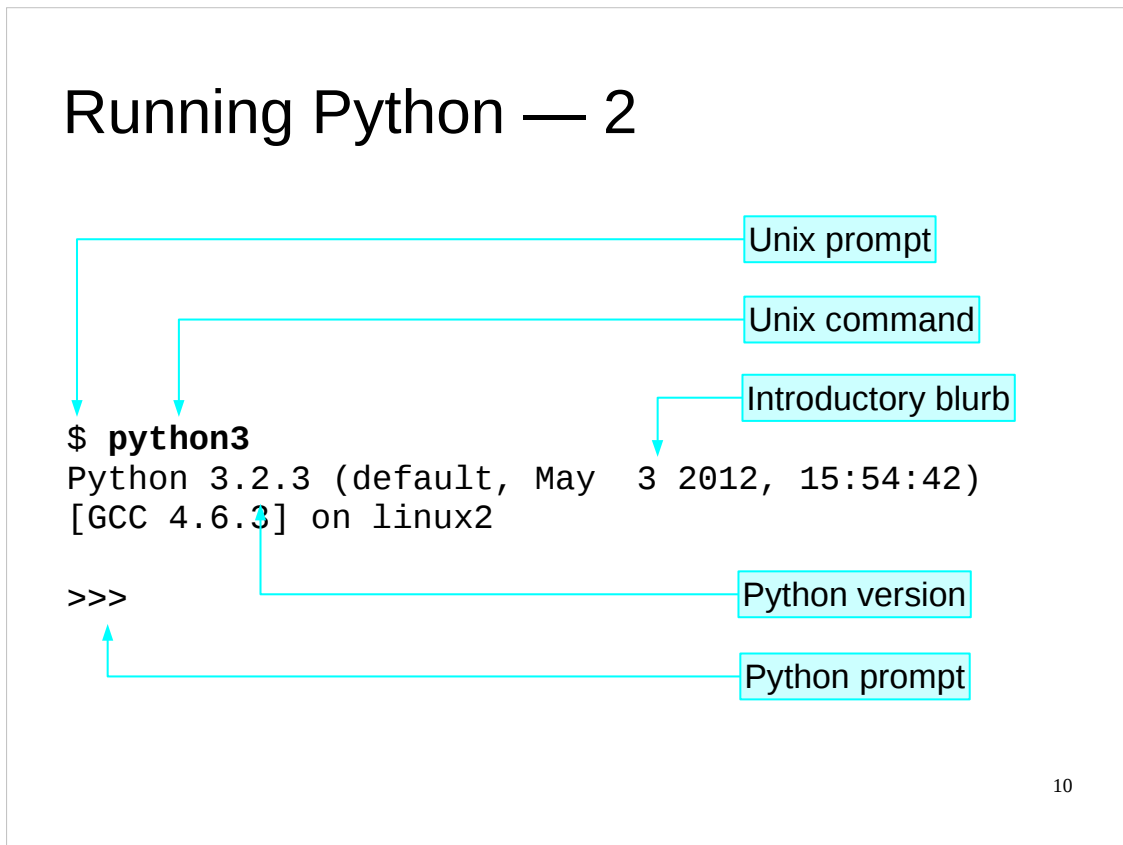
Running Python — 1
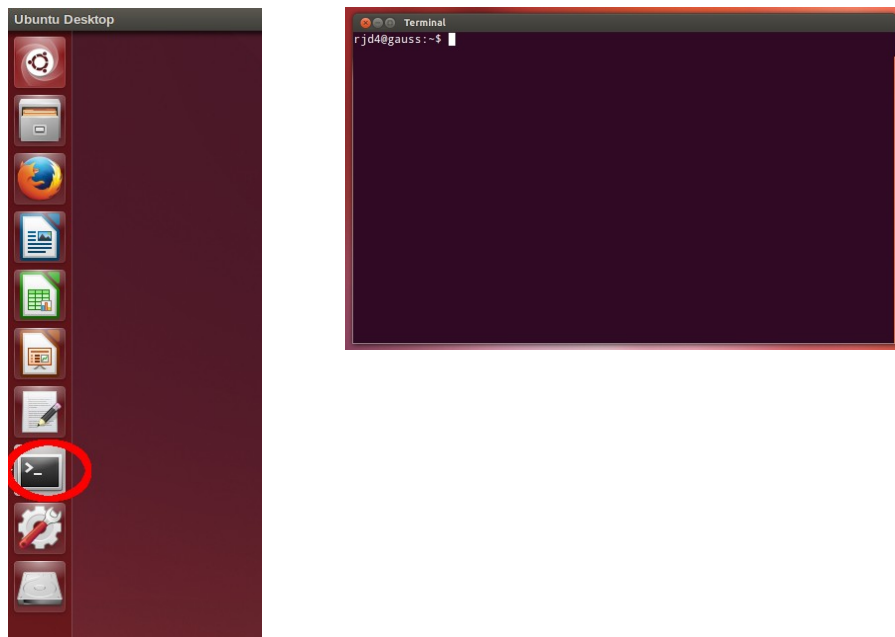
We are going to use Python from the command line either directly or indirectly.

So, first I need a Unix command line. I will get that from the GUI by clicking on the terminal icon in the desktop application bar.

Running Python — 2

Unix prompt

Unix command

Introductory blurb

```
$ python3
Python 3.2.3 (default, May  3 2012, 15:54:42)
[GCC 4.6.3] on linux2

>>>
```

Python version

Python prompt

10

Now, the Unix interpreter prompts you to give it a Unix command with a short bit of text that ends with a dollar. In the slides this will be represented simply as a dollar.

This is a Unix prompt asking for a Unix command.

The Unix command we are going to give is "python3". Please note that trailing "3". The command "python" gives you either Python 2 or Python 3 depending on what system you are on. With this command we are insisting on getting a version of Python 3.

The Python interpreter then runs, starting with a couple of lines of blurb. In particular it identifies the specific version of Python it is running. (3.2.3 in this slide.)

Then it gives a prompt of its own, three "greater than" characters. The Python 3 program is now running and it is prompting us to give a Python command.
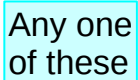
You cannot give a Unix command at a Python prompt (or *vice versa*).

Quitting Python

```
>>> exit()


>>> quit()                   Any one
                             of these

>>> Ctrl + D
```

There are various ways to quit interactive Python. There are two commands which are equivalent for our purposes: `quit()` and `exit()`, but the simplest is the key sequence [Ctrl]+[D].

A first Python command

Python prompt

Python command

```
>>> print('Hello, world!')

Hello, world!

>>>
```

Output

Python prompt

There is a tradition that the first program you ever run in any language generates the output "Hello, world!".

I see no reason to buck tradition. Welcome to your first Python command; we are going to output "Hello, world!".
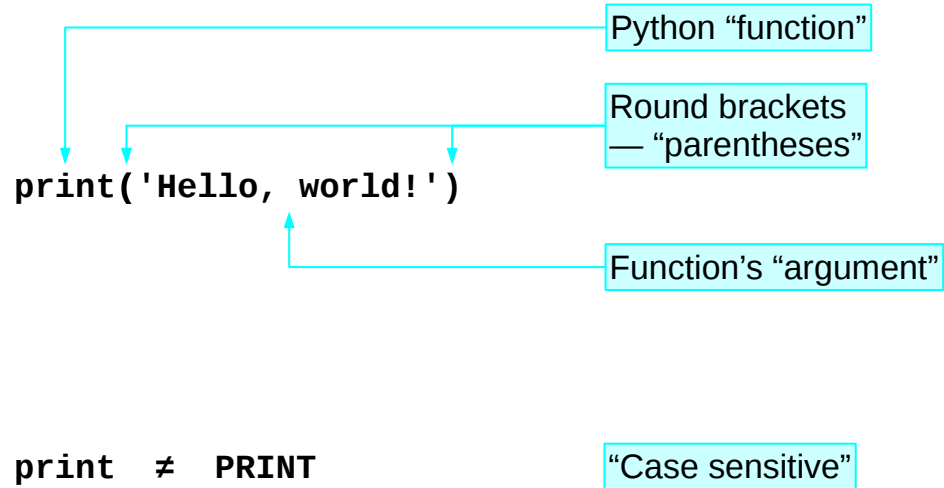
We type this command at the Python prompt. The convention in these slides is that the typewriter text in bold face is what you type and the text in regular face is what the computer prints.

We type "print" followed by an opening round brackets and the text "Hello, world!" surrounded by single quotes, ending with a closing round bracket and hitting the Return key, [↵], to indicate that we are done with that line of instruction.

The computer responds by outputting "Hello, world!" without the quotes.

Once it has done that it prompts us again asking for another Python command with another Python prompt, ">>>".

Python commands


```
print('Hello, world!')
```

Python "function"

Round brackets — "parentheses"

Function's "argument"

```
print  ≠  PRINT
```

"Case sensitive"

This is our first Python "function". A function takes some input, does something with it and (optionally) returns a value. The nomenclature derives from the mathematics of functions, but we don't need to fixate on the mathematical underpinnings of computer science in this course.

Our function in this case is "`print`" and the command necessarily starts with the name of the function.

The inputs to the function are called its "arguments" and follow the function inside round brackets ("parentheses").

In this case there is a single argument, the text to print.

Note that Python, as with many but not all programming languages, is "case sensitive". The word "`print`" is not the same as "`Print`" or "`PRINT`".

Python text

Quotation marks

`'Hello, world!'`
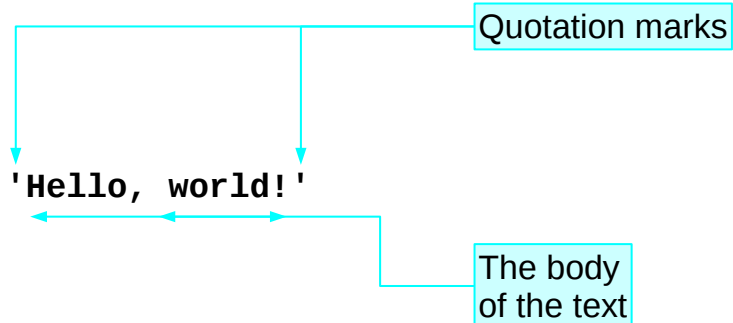
The body
of the text

! The quotes are not
part of the text itself.

The text itself is presented within single quotation marks. (We will discuss the choice of quotation marks later.)

The body of the text comes within the quotes.

The quotes are not part of the text; they merely indicate to the Python interpreter that "hey, this is text!"

Recall that the the printed output does not have quotes.

Quotes?

```
print            ─────────────►  Command
```

```
'print' ─────────────►  Text
```

So what do the quotes "do"?

If there are no quotes then Python will try to interpret the letters as something it should know about. With the quotes Python simply interprets it as literal text.

For example, without quotes the string of characters `p-r-i-n-t` are a command; with quotes they are the text to be printed.
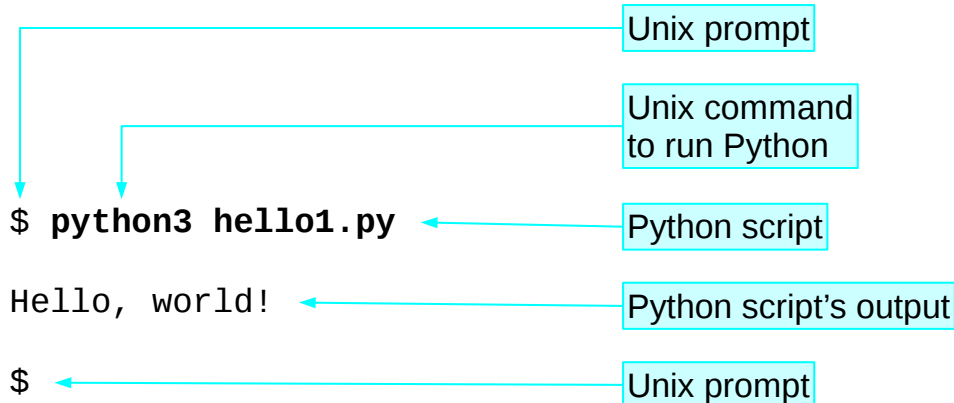
Python scripts

File in home directory

Run from *Unix* prompt

```
print('Hello, world!')
```

hello1.py

Unix prompt

Unix command
to run Python

```
$ python3 hello1.py
```
Python script

```
Hello, world!
```
Python script's output

```
$
```
Unix prompt

So we understand the "hello, world" command and how to run it from an interactive Python. But serious Python programs can't be typed in live; they need to be kept in a file and Python needs to be directed to run the commands from that file.

These files are called "scripts" and we are now going to look at the Python script version of "hello, world".

In your home directories we have put a file called "hello1.py". It is conventional that Python scripts have file names ending with a ".py" suffix. Some tools actually require it. We will follow this convention and you should too.

This contains exactly the same as we were typing manually: a single line with the print command on it.

We are going to make Python run the instructions out of the script. We call this "running the script".

Scripts are run from the Unix command line. We issue the Unix command "python3" to execute Python again, but this time we add an extra word: the name of the script, "hello1.py".

When it runs commands from a script, python doesn't bother with the lines of blurb and as soon as it has run the commands (hence the output) it exists immediately, returning control to the Unix environment, so we get a Unix prompt back.

Editing Python scripts — 1
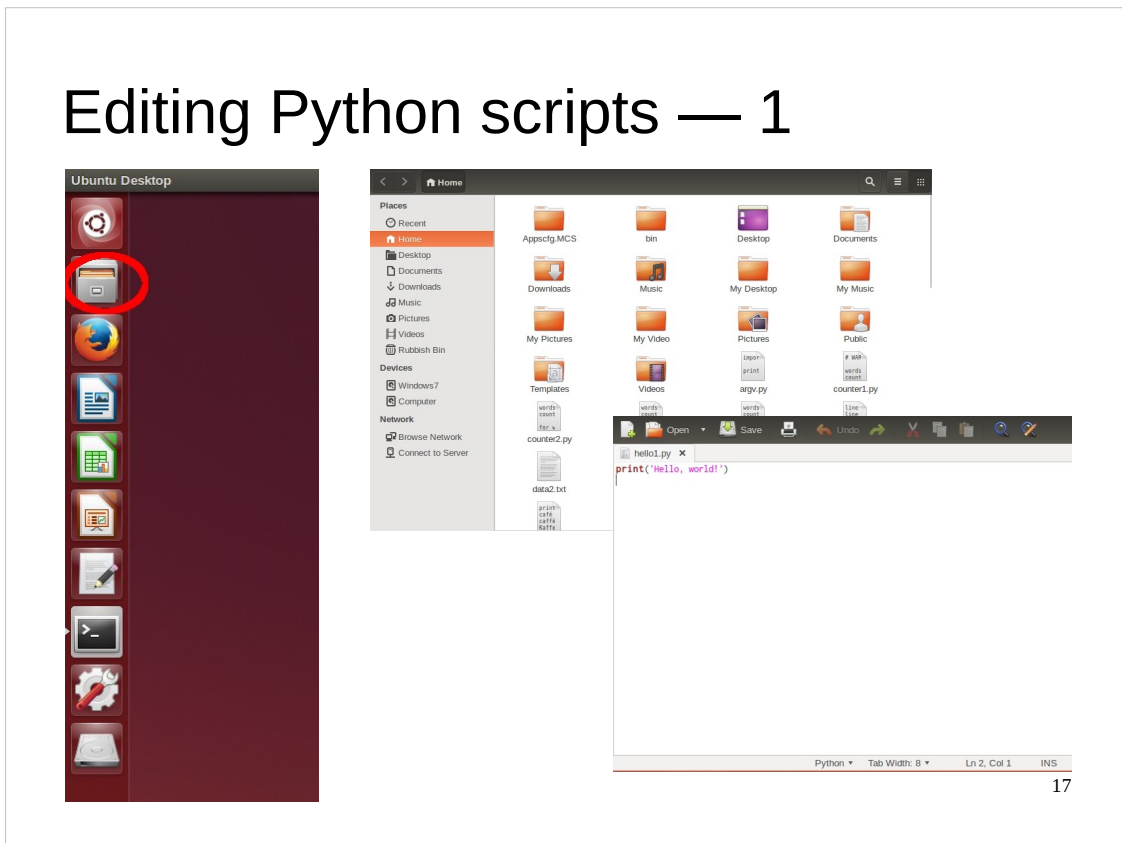
To edit scripts we will need a plain text editor. For the purposes of this course we will use an editor called "`gedit`". You are welcome to use any text editor you are comfortable with (e.g. `vi` or `emacs`).

If a script already exists then we can launch the file browser and simply double-click on its icon.

This will launch the appropriate application. For a Python (text) script this is the text editor (and not a Python instance to run the script).

Editing Python scripts — 2
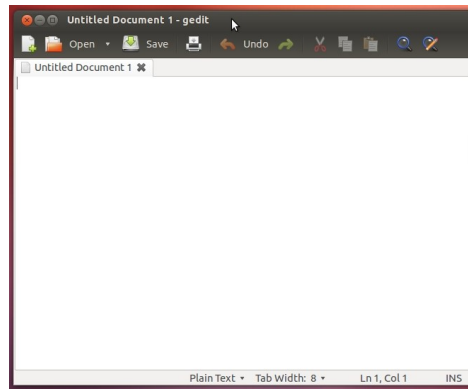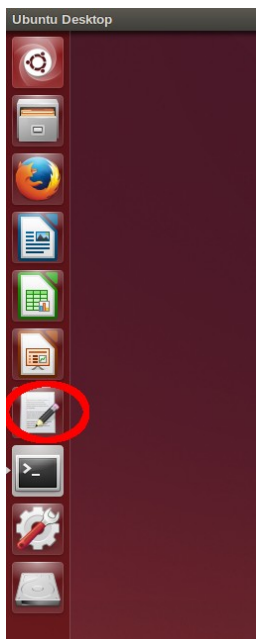
18

If the file does not already exist then click on the text editor icon in the dock to launch it with no content.

Just as with the terminal you can get this from the dock on the left hand side of the screen.

Progress

Interactive Python

Python scripts

`print()` command

Simple Python text

19

Exercise 1

1. Print "Goodbye, cruel world!" from interactive Python.

2. Edit `exercise1.py` to print the same text.

3. Run the modified `exercise1.py` script.

❣ Please ask if you have questions.

2 minutes

20

During this course there will be some "lightning exercises". These are very quick exercises just to check that you have understood what's been covered in the course up to that point.

Here is your first.

First, make sure you can print text from interactive Python and quit it afterwards.

Second, edit the `exercise1.py` script and run the edited version with the different output.

This is really a test of whether you can get the basic tools running. *Please* ask if you have any problems!

A little more text

Full "Unicode" support

```
print('ħəɫʊo, ω☺ɲ∂‼')
```

hello2.py

www.unicode.org/charts/

Now let's look at a slightly different script just to see what Python can do.

Python 3 has excellent support for fully international text. (So did Python 2 but it was concealed.)

Python 3 supports what is called the "Unicode" standard, a standard designed to allow for characters from almost every language in the world. If you are interested in international text you need to know about the Unicode standard. The URL shown will introduce you to the wide range of characters supported.

The example in the slide contains the following characters:

ħ	PLANCK'S CONSTANT DIVIDED BY TWO PI
ə	CYRILLIC SMALL LETTER E
ɫ	LATIN SMALL LETTER L WITH BAR
	CHEROKEE LETTER DA
ʊ	ETHIOPIC SYLLABLE PHARYNGEAL A
ω	GREEK SMALL LETTER OMEGA
☺	WHITE SMILING FACE
ɲ	ARMENIAN SMALL LETTER REH
	COPTIC SMALL LETTER LAUDA
∂	PARTIAL DIFFERENTIAL
‼	DOUBLE EXCLAMATION MARK

Getting characters

AltGr + Shift + #   g   Linux

Character Selector

ğ

"LATIN SMALL
LETTER G
WITH BREVE"

\u011f
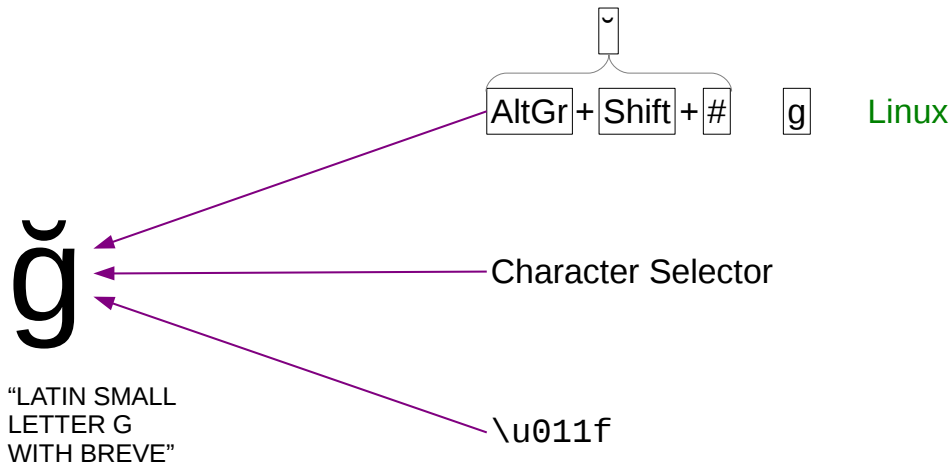
I don't want to get too distracted by international characters, but I ought to mention that the hardest part of using them in Python is typically getting them into Python in the first place.

There are three "easy" ways.

There are key combinations that generate special characters. On Linux, for example, the combination of the three keys [AltGr], [Shift], and [#] set up the breve accent to be applied to the next key pressed.

Perhaps easier is the "Character Selector" application. This runs like a free-standing "insert special character" function from a word processor. You can select a character from it, copy it to the clipboard and paste it into any document you want.

Finally, Python supports the idea of "Unicode codes". The two characters "\u" followed by the hexadecimal (base 16) code for the character in the Unicode tables will represent that character. You have all memorized your code tables, haven't you?

Text: a "string" of characters
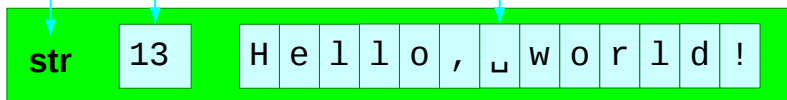
```
>>> type('Hello, world!')

<class 'str'>
```

A string of characters

Class: string

Length: 13

Letters

| str | 13 | | H | e | l | l | o | , | ␣ | w | o | r | l | d | ! |

23

We will quickly look at how Python stores text, because it will give us an introduction to how Python stores *everything*.

Every object in Python has a "type" (also known as a "class").

The type for text is called "str". This is short for "string of characters" and is the conventional computing name for text. We typically call them "strings".

Internally, Python allocates a chunk of computer memory to store our text. It stores certain items together to do this. First it records that the object is a string, because that will determine how memory is allocated subsequently. Then it records how long the string is. Then it records the text itself.

Text: "behind the scenes"

str	13		72	101	108	108	111	44	32		100	33
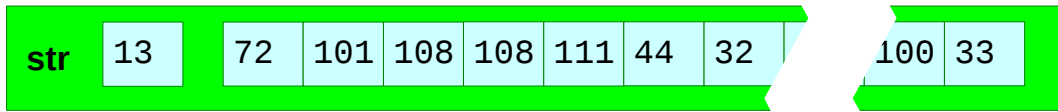
```
>>> '\u011f'

'ğ'

>>> ord('ğ')

287

>>> chr(287)

'ğ'
```

$011f_{16}$

$\updownarrow$

287_{10}

$\updownarrow$

ğ

24

In these slides I'm going to represent the stored text as characters because that's easier to read. In reality, all computers can store are numbers. Every character has a number associated with it. You can get the number corresponding to any character by using the `ord()` function and you can get the character corresponding to any number with the `chr()` function.

Mathematical note:

The subscript 10 and 16 indicate the "base" of the numbers.

Adding strings together: +

"Concatenation"

```
print('Hello, ' + 'world!')
```
hello3.py

```
>>> 'Hello, '  +  'world!'

'Hello, world!'

>>>
```

Now let's do something with strings.

If we 'add' two strings together Python joins them together to form a longer string.

Python actually permits you to omit the "+". Don't do this.

Pure concatenation

```
>>> 'Hello,␣'  +  'world!'

'Hello, world!'


>>> 'Hello,'  +  '␣world!'

'Hello, world!'


>>> 'Hello,'  +  'world!'

'Hello,world!'
```

Only simple concatenation

No spaces added automatically.

This joining together is very simple. If you want words split by a space you have to put the space in.

Single & double quotes

```
>>> 'Hello, world!'          Single quotes

'Hello, world!'              Single quotes


>>> "Hello, world!"          Double quotes

'Hello, world!'              Single quotes
```
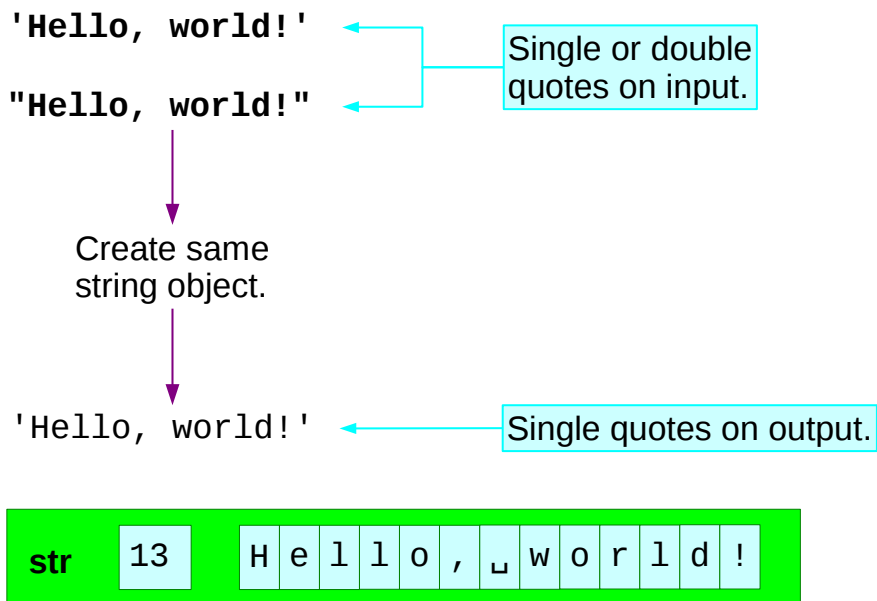
27

It doesn't matter whether we write our strings with single or double quotes (so long as they match at the two ends). Python simply notes that we are defining a string.

Python strings: input & output

'Hello, world!' ← Single or double quotes on input.

"Hello, world!"

↓

Create same string object.

↓

'Hello, world!' ← Single quotes on output.

str	13		H	e	l	l	o	,	␣	w	o	r	l	d	!

28

Internally there are no quotes, just a record that the object is text.

When Python comes to display the string and declares "this is text" itself it uses single quotes.

Uses of single & double quotes

```
>>> print('He said "hello" to her.')

He said "hello" to her.


>>> print("He said 'hello' to her.")

He said 'hello' to her.
```

29

Having two sorts of quotes can be useful in certain circumstances. If you want the text itself to include quotes of one type you can define it surrounded by the other type.

Why we need different quotes

```
>>> print('He said 'hello' to her.')

File "<stdin>", line 1
  print('He said 'hello' to her.')
                       ^
SyntaxError: invalid syntax
```

You must mix the quotes like that. If you do not then Python will be unable to make sense of the command.

We will look at Python's error messages in more detail later.

Adding arbitrary quotes

```
>>> print('He said \'hello\' to her.')

He said 'hello' to her.
```

\' ————————▶ '   Just an ordinary character.

\" ————————▶ "   "Escaping"

str	23	H	e	␣	s	a	i	d	␣	'	h	e	l	l	o	'	␣	t	o	␣	h	e	r	.

There is a more general solution to the "quotes within quotes" problem.

Preceding each quote within the body of the text signals to Python that this is just an ordinary quote character and should not be treated specially.

Note that what is encoded in the string is a single character. The backslash is a signal to the Python interpreter as its constructs the string. Once the string is constructed, with quotes in it, the backslash's work is done.

This process of flagging a character to be treated differently than normal is called "escaping" the character.

Putting line breaks in text

```
Hello,
world!
```

```
>>> print('Hello, ↵
world')
```

```
>>> print('Hello, ↵
File "<stdin>", line 1
   print('Hello,
                ^
SyntaxError: EOL while
scanning string literal
```
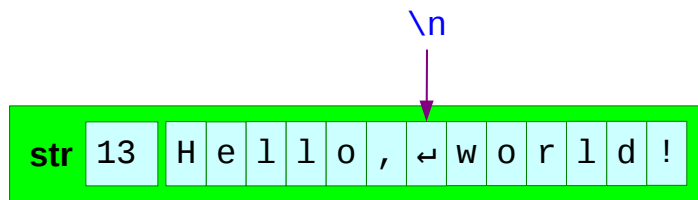
"EOL": End Of Line

We will follow the theme of "inserting awkward characters into strings" by looking at line breaks.

We cannot insert a line break by hitting the [↵] key. This signals to Python that it should process the line so far and Python cannot; it is incomplete.

Inserting "special" characters

```
>>> print('Hello,\nworld!')
Hello,
world!
```
Treated as a new line.

\n

Converted into a *single* character.

str 13 | H | e | l | l | o | , | ↵ | w | o | r | l | d | !

```
>>> len('Hello,\nworld!')
13
```
len() function: gives the length of the object

Again, the backslash character comes to our rescue.

If we create a string with the sequence "\n" then Python interprets this as the *single* character ↵.

Python can tell us exactly how many characters there are in a string. The len() function tells us the length of the string in characters. There are 13 characters in the string created by 'Hello,\nworld!'. The quotes are not part of the text and the \n becomes a single character.

The backslash

Special ⟶ Ordinary   \' ⟶ '

                                   \" ⟶ "

Ordinary ⟶ Special   \n ⟶ ↵

                                   \t ⟶ ⇥

We have used backslash again, this time for a slightly different result. Backslash before a character with special significance, such as the quote character, makes the character "ordinary". Used before an ordinary character, such as "n", it produces something "special".

Only a few ordinary characters have special characters associated with them but the two most commonly useful are these:

\n   ↵   new line

\t   ⇥   tab stop

\n: unwieldy for long text

```
'SQUIRE TRELAWNEY, Dr. Livesey, and the\n
rest of these gentlemen having asked me\n
to write down the whole particulars\nabou
t Treasure Island, from the\nbeginning to
 the end, keeping nothing\nback but the b
earings of the island,\nand that only bec
ause there is still\ntreasure not yet lif
ted, I take up my\npen in the year of gra
ce 17__ and go\nback to the time when my
father kept\nthe Admiral Benbow inn and t
he brown\nold seaman with the sabre cut f
irst\ntook up his lodging under our roof.'
```

Single line

The "\n" trick is useful for the occasional new line. It is no use for long texts where we want to control the formatting ourselves.

Special input method for long text

```
'''SQUIRE TRELAWNEY, Dr. Livesey, and the
rest of these gentlemen having asked me
to write down the whole particulars
about Treasure Island, from the
beginning to the end, keeping nothing
back but the bearings of the island,
and that only because there is still
treasure not yet lifted, I take up my
pen in the year of grace 17__ and go
back to the time when my father kept
the Admiral Benbow inn and the brown
old seaman with the sabre cut first
took up his lodging under our roof.'''
```

Triple quotes

Multiple lines

Python has a special trick precisely for convenient definition of long, multi-line text.

If you start the text with a "triple quote" then the special treatment of hitting the [↵] key is turned off. This lets you enter text "free form" with natural line breaks.

The triple quote is three quote characters with no spaces between them. The quote character used can be either one but the triple use at one end must match the one used at the other end.

Python's "secondary" prompt

```
>>> '''Hello,
... world'''
```

Python asking for more of the same command.

The triple quote lets us see another Python feature. If we type a long string raw then after we hit ↵ we see Python's "secondary prompt". The three dots indicate that Python is expecting more input before it will process what it has in hand.

It's still just text!

```
>>> 'Hello,\nworld!'

'Hello\nworld'
```

Python uses \n to represent line breaks in strings.

```
>>> '''Hello,
... world!'''

'Hello\nworld'
```

Exactly the same!

It is also important to note that triple quotes are just a trick for input. The text object created is still a standard Python string. It has no memory of how it was created.

Also note that when Python is representing the content of a string object (as opposed to printing it) it displays new lines as "\n".

Your choice of input quotes:

Four inputs:

      `'Hello,\nworld!'`   `"Hello,\nworld!"`

      `'''Hello,`   `"""Hello,`
      `world!'''`   `world!"""`

Same result:

str	13	H	e	l	l	o	,	↵	w	o	r	l	d	!

39

We have now seen four different ways to create a string with an embedded new line. They all produce the same string object.

Progress

International text

`print()`

Concatenation of strings

Special characters

Long strings

40

Exercise 2

1. Replace XXXX in `exercise2.py` so it prints the following text (with the line breaks) and then run the script.

   coffee
   café
   caffè
   Kaffee

| é | \u00e8 | AltGr + ; | e |
| è | \u00e9 | AltGr + # | e |

3 minutes

There is more than one way to do this.

You can get the line breaks with \n in a single-quoted string or with literal line breaks in a triple-quoted string. An alternative, but not in keeping with the exercise, is to have four `print()` statements.

You can get the accented characters by using the `\u` sequences or you can type them in literally with the keyboard combinations shown. (Linux only)

Attaching names to values

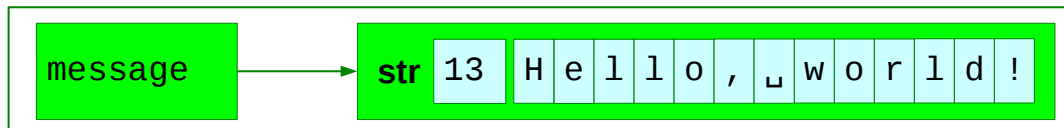"variables"

```
message = 'Hello, world!'
print(message)
```

hello3.py

```
>>> message='Hello, world!'

>>> message

'Hello, world!'

>>> type(message)

<class 'str'>
```

```
message  →  str  13  H e l l o ,  ␣ w o r l d !
```

Now we will move on to a serious issue in learning any computing language: how to handle names for values.

Compare the two scripts `hello1.py` and `hello4.py`. They both do exactly the same thing.

We can enter the text of `hello4.py` manually if we want using interactive Python, it will work equally well there.

The first line of `hello4.py` creates the string 'Hello, world!' but instead of printing it out directly the way that `hello1.py` does, it attaches a name, "`message`", to it.

The second line runs the `print()` function, but instead of a literal string as its argument it has this name instead. Now the name has no quotes around it, and as I said earlier this means that Python tries to interpret it as something it should do something with. What it does is to look up the name and substitute in the attached value.

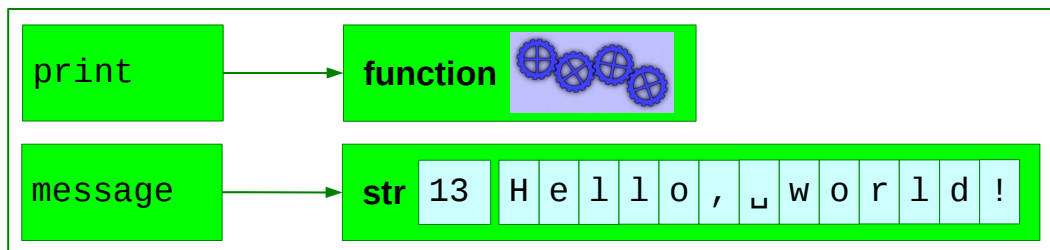Whenever the name is used, Python will look it up and substitute in the attached value.

Attaching names to values

```
message = 'Hello, world!'
print(message)
```

hello4.py

```
>>> type(print)

<class 'builtin_function_or_method'>
```


Both "`print`" and "`message`" are the same this way. Both are names attached to Python objects. "`print`" is attached to a chunk of memory containing the definition of a function and "`message`" is attached to a chunk of memory containing the text.

Reading some text into a script

```
message = input('Yes?␣')
print(message)
```

$ **python3 input1.py** ←——————————————— input1.py

                       ←——————————————— input('Yes?␣')

Yes? **Boo!** ——————————————→ message = …

Boo! ←——————————————— print(message)

Now that we know how to attach names to values we can start receiving input from the user of our script.

For this we will use the cunningly named "`input()`" function.

This function takes some (typically short) text as its argument. It prints this text as a prompt and then waits for the user to type something back (and press [↵]. It then returns whatever the user typed (without the [↵]) as its value.

We can use this function on the right hand side of an assignment.

Recall that the assignment completely evaluates the right hand side first. This means that it has to evaluate the `input()` function, so it prompts the user for input and evaluates to whatever it was that the user typed.

Then the left hand side is processed and the name "`message`" is attached to this value.

We can then print this input text by using the attached name.

Can't read numbers directly!

```
$ python3 input2.py

N? 10
```

```
number = input('N? ')
print(number + 1)
```

input2.py

```
Traceback (most recent call last):
  File "input2.py", line 2, in <module>
    print(number + 1)
TypeError:
Can't convert 'int' object
to str implicitly
```

string   integer

In the previous example script `input1.py` we simply took what we were given by `input()` and printed it. The `print()` function is a flexible beast; it can cope with almost anything.

The script `hello2.py` attempts to take what is given and do arithmetic with it, namely add 1 to it. It fails, even though we type a number at `input()`'s prompt.

This also gives us an error message and it's time to investigate Python's error messages in more detail.

The first (the "trace back") tells us where the error was. It was on line 2 of the file `input2.py`. It also tells us what was on the line. Recall that with syntax errors it also pointed out where in the line it realized something was going wrong.

The second part tells us *what* the error was: we tried to add a string (text) and an integer (a number). More precisely, Python couldn't convert the things we were adding together into things that we could add.
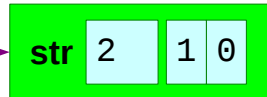
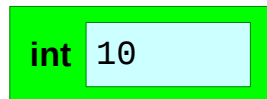# `input()`: strings only

```
$ python3 input2.py

N? 10
```

```
number = input('N? ')
print(number + 1)
```
✗

input2.py

`input('N? ')` ⟶ **str** `2` `1` `0`

≠ **int** `10`

The problem is that the `input()` function always returns a string and the string "character 1 followed by character 0" is not the same as the integer ten.

We will need to convert from the string to the integer explicitly.

Some more types

```
>>> type('Hello, world!')

<class 'str'>
```
string of characters

```
>>> type(42)

<class 'int'>
```
integer

```
>>> type(3.14159)

<class 'float'>
```
floating point number

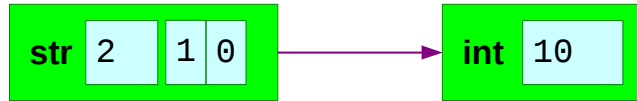To date we have seen only two types: "`str`" and "`builtin_function_or_method`". Here are some more.

Integers (whole numbers) are a type called "`int`".

Floating point numbers (how computers approximate real numbers) are a type called "`float`".
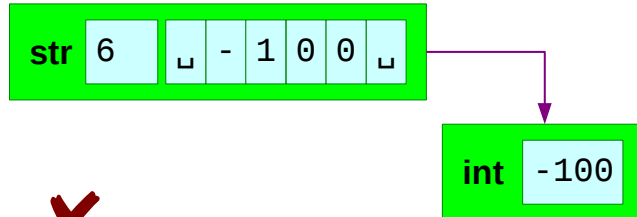
The `input()` function gave is a "`str`". We want an "`int`".

Converting text to integers

```
>>> int('10')
10
```



```
>>> int(' -100 ')
-100
```



```
>>> int('100-10')

ValueError:
invalid literal for int() with base 10: '100-10'
```

There is a function — also called "int()" — that converts the textual representation of an integer into a genuine integer.

It copes with extraneous spaces and other junk around the integer but it does not parse general expressions. It will take the textual form of a number, but that's it.

Converting text to floats

```
>>> float('10.0')
```
'10.0' is a string

```
10.0
```
10.0 is a floating point number

```
>>> float(' 10. ')

10.0
```

There is a similar function called `float()` which creates floating point numbers.

Converting between ints and floats

```
>>> float(10)

10.0


>>> int(10.9)

10

>>> int(-10.9)

-10
```

Truncates fractional part

The functions can take more than just strings, though. They can take other numbers, for example. Note that the `int()` function truncates floating point numbers.

Converting into text

```
>>> str(10)            integer ——► string

'10'


>>> str(10.000)        float ——► string

'10.0'
```

There is also a `str()` function for turning things into strings.

Converting between types

`int()`   anything ⟶ integer

`float()`   anything ⟶ float

`str()`   anything ⟶ string

Functions named after the type they convert *into*.

In general there is a function for each type that converts whatever it can into that type.

Reading numbers into a script

```
text = input('N?␣')
number = int(text)
print(number + 1)
```

```
$ python3 input3.py

N? 10

11
```

So finally we can see what we have to do to make our failing script work: we need to add a type conversion line.

# Stepping through our script — 1

str | 3 | | N | ? | ␣

```
text = input('N?␣')
number = int(text)
print(number + 1)
```

We can step through the script one element at a time to see the totality of what we have written.

Starting on the right hand side of the assignment operator (as we always do), initially we have the literal string which forms the body of the prompt.
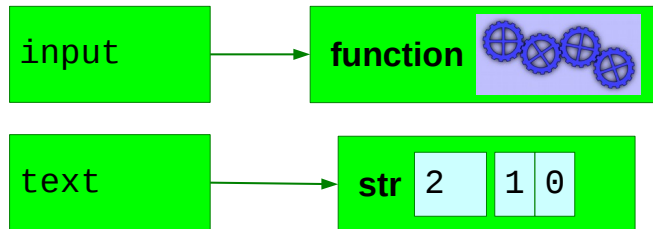
Stepping through our script — 2

str | 3 | | N | ? | ␣

```
text = input('N?␣')
number = int(text)
print(number + 1)
```

input ⟶ **function** ⚙⚙⚙

str | 2 | | 1 | 0 ← NB: text, not number
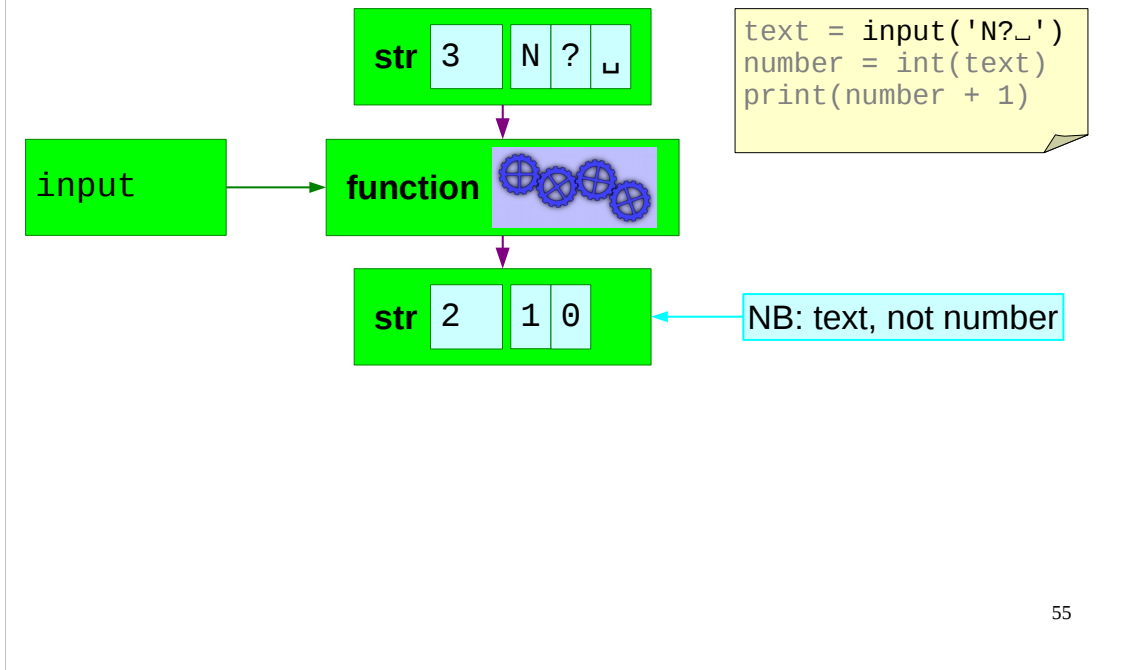
Looking at the rest of the right hand side of the line we see that this is used by the `input` function. Its name is looked up and the string object "N? " is passed to it. The function returns (evaluates to) a second text object "10". Note that this is a string object (text) and not an integer (number).

Stepping through our script — 3

```
text = input('N? ')
number = int(text)
print(number + 1)
```
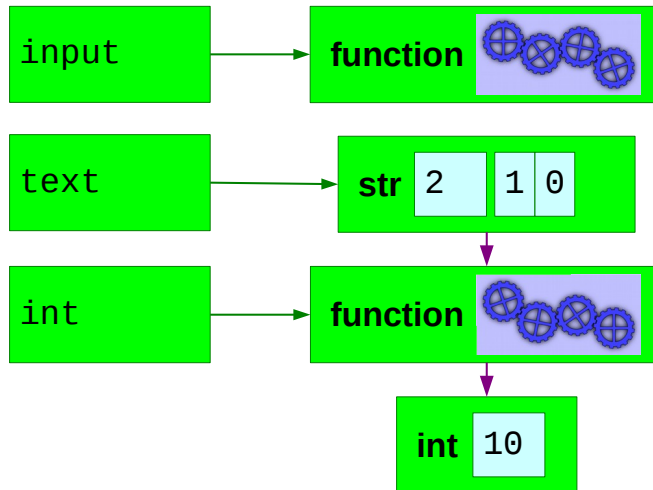
input → function

text → str 2 1 0

Now that Python has completd the evaluation of the right hand side its attention moves to the left hand side. This identifies the name, `text`, which is attached to this string value.

Stepping through our script — 4

```
text = input('N? ')
number = int(text)
print(number + 1)
```

input ────→ **function** 🔩🔩🔩

text ────→ **str** 2 1 0

int ────→ **function** 🔩🔩🔩

int 10

57

Moving to the right hand side of the second line, the name `text` is looked up to get the string "`10`" and the name `int` is looked up to give the function that converts strings into integers. The string is passed to the function as its argument and the function returns an integer value, `10`.

This completes the evaluation of the right hand side.

Stepping through our script — 5

```
text = input('N?␣')
number = int(text)
print(number + 1)
```

input → **function** [gears]

text → **str** 2 1 0

int → **function** [gears]
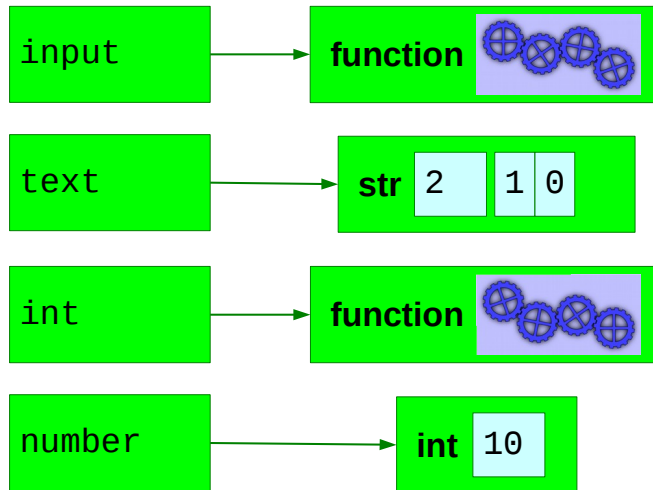
number → **int** 10

58

Moving now to the left hand side, Python sees the name `number` and attaches it to the integer value given by the right hand side.
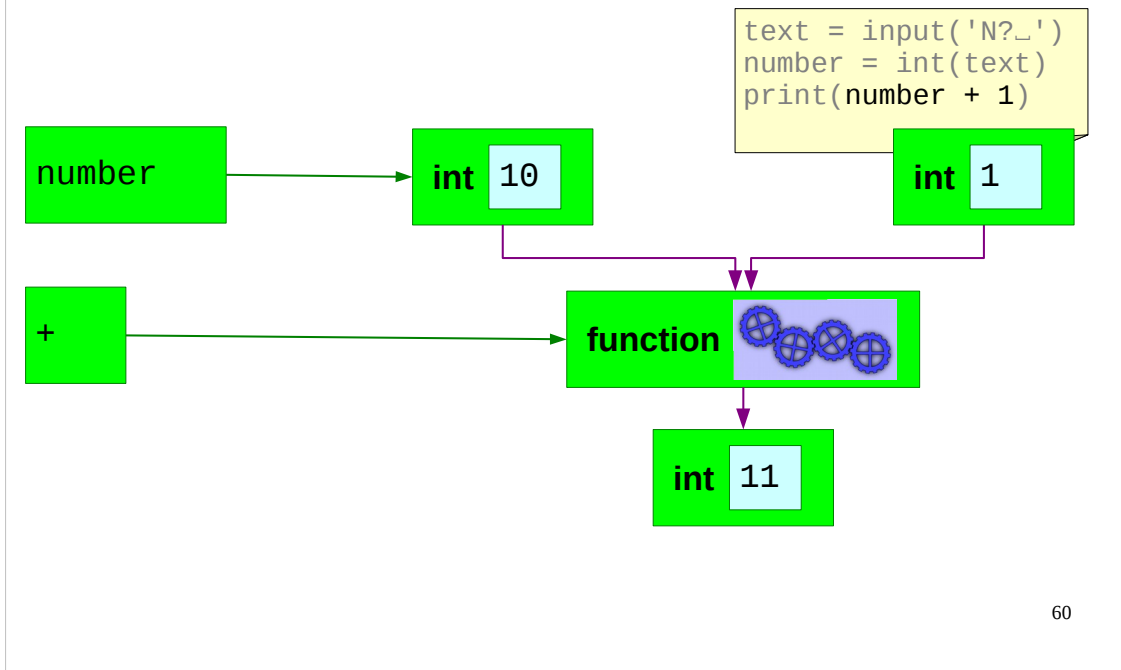
Scrolling up, because slides are never big enough…

Stepping through our script — 7

```
text = input('N?␣')
number = int(text)
print(number + 1)
```

number → **int** `10`

int `1`

+ → **function**

int `11`

60

The third line contains the expression `number+1`. Python looks up the name `number` to give an integer `10`, looks up the name + to give a function, creates an integer `1` directly and calls the function to add the numbers together. This yields a result of integer `11`.

Stepping through our script — 6

```
text = input('N?␣')
number = int(text)
print(number + 1)
```

number ——→ **int** 10

int 11
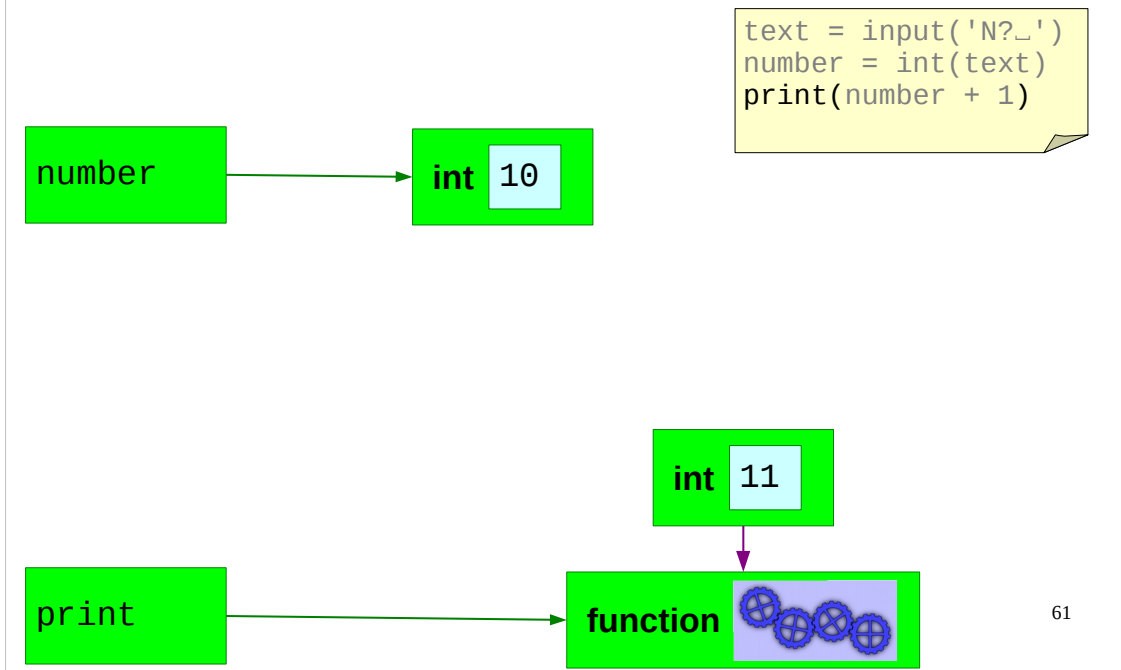
print ——————→ **function**

Finally, Python looks up the name `print` to give a function and passes it the freshly created value of integer 11 as an argument.

Progress

Names ⟶ Values

$name$ = $value$

Types

strings

integers

floating point numbers

Reading in text

input($prompt$)

Type conversions

str() int() float()

62

Exercise 3

Replace the two XXXX in exercise3.py to do the following:

1. Prompt the user with the text "How much?␣".

2. Convert the user's answer to a floating point number.

3. Print 2.5 plus that number.

3 minutes

Integers

$$\mathbb{Z} \ \{... \ -2, -1, 0, 1, 2, 3, 4 ...\}$$

Now that we have some rudimentary understanding of Python it's time to dive in a little deeper to the three types we have met so far.

We are going to start with the whole numbers, "integers" in technical language.

Mathematical note:

The fancy Z is the mathematical symbol for the integers based on the German word *Zahlen*.

Integer addition & subtraction

```
>>> 20+5
25


>>> 20 - 5
15
```

Spaces around the
operator don't matter.

"No surprises"

We can start our handling of integers with some very basic arithmetic.
Note that spaces around the plus and minus character are ignored.
Adding or subtracting two integers simply gives a third integer.

Integer multiplication

There is no "×" on the keyboard.

Linux:
AltGr + Shift + ,

Use "*" instead

```
>>> 20 * 5
100
```

Still no surprises

The arithmetical operations addition and subtraction have their usual mathematical symbols reflected on the standard keyboard. We have a plus sign and a minus sign (actually a "hyphen") character and we use them.

There is no multiplication symbol on the standard keyboard. You can generate it as one of the octopus-friendly key combinations, but it's not a simple key.

Instead, the computing profession has settled on using the asterisk ("*") to represent multiplication. On your keyboards this is [Shift]+[8].

Multiplying two integers gives a third integer.

Integer division

There is no "÷" on the keyboard.

Use "/" instead

Linux:
AltGr + Shift + .

```
>>> 20 / 5
4.0
```
This is a floating point number!

Surprise!

There is no division symbol on the keyboard without holding three keys down at the same time. Again a convention has arisen to use the forward slash character (strictly called a "solidus") for division.

So far there have been no surprises in Python's integer arithmetic. That changes with division.

Not all integer division can be achieved precisely. You cannot divide 3 into 5 exactly. Because of this Python 3 always returns a type of number capable of representing fractional values (a floating point number) even when the division would have been exact.

Integer division gives floats !

Fractions ———————➤ Floats sometimes

Consistency ———————➤ Floats **always**

```
>>> 20 / 40
0.5

>>> 20 / 30
0.6666666666666666
```

The designers of Python decided that consistency of output was important and therefore because it might sometimes need to use a float it should always use a float.

Note that even floating point numbers cannot exactly represent all fractions. ½ can be precisely represented but ⅔ cannot. We will return to the imprecision of floating point numbers when we look at them in detail.

(If you really want to stick to integers then Python 3 offers the "`//`" operator which returns an integer answer, rounded strictly down in case of fractional answers.)

Integer powers

There is no "4^2" on the keyboard.

Use "$**$" instead

```
>>> 4 ** 2
16

>>> 4* *2
SyntaxError: invalid syntax
```

Spaces *around* the operator don't matter.

Spaces *in* the operator do!

Just as there is no mathematical symbol on the keyboard for multiplication and division, there is no symbol at all for raising to powers. Mathematically we represent it by superscripting the power after the number being raised.

We can't do this on the keyboard so instead we cheat and invent our own symbol for the operation.

Computing has split for this operation. Some languages use the circumflex accent ("^") and others, including Python, use a double asterisk, "**".

Note that while spaces around the operator are ignored you can't split the two asterisks.

Integer remainders

e.g. Is a number even or odd?

Use "%"

```
>>> 4 % 2
0

>>> 5 % 2
1

>>> -5 % 2
1
```

Remainder is always non-negative

There's one last integer arithmetic operator we will use once in a while.

Another way to look at division is to ask what the remainder is after a division. Python represents this concept by using the percent sign between to numbers to represent the remainder when the first number is divided by the second.

We will use it for one purpose only in this course: to determine if a number is even or odd. If a number's remainder when divided by 2 is 0 then it's even and if the remainder is 1 then it's odd.

How big can a Python integer be?

```
>>> 2**2
4

>>> 4**2
16

>>> 16**2
256

>>> 256**2
65536

>>> 65536**2
4294967296
```

Now we will look at the numbers themselves. We can ask the question "how big can an integer be?" Mathematically, of course, there is no limit. In a computer there are always limits. Each computer has only a finite amount of memory to store information so there has to be a limit. We will see that Python has no limits in principle and it is only the technical limit of the computer that can restrict us. In practice this is never an issue for the size of integers.

We will experiment with large integers by repeated squaring. We start with a 2 and keep squaring.

How big can a Python integer be?

```
>>> 4294967296**2
18446744073709551616

>>> 18446744073709551616**2
340282366920938463463374607431768211456

>>> 340282366920938463463374607431768211456**2
11579208923731619542357098500868790785326998466564056403945758400791312963993 6

>>> 115792089237316195423570985008687907853269
9846656405640394575840079131296399 36**2
13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569 94643364900608409 6
```

72

Python takes it in its stride and happily carries on.

How big can a Python integer be?

10443888814131525066917527107166243825799642490473837803842334832839
53907971557456848826811934997558340890106714439262837987573438185793
60726323608785136527794595697654370999834036159013438371831442807001
18559462263763188393977127456723346843445866174968079087058037040712
84048740118609114467977783598029006686938976881787785946905630190260
94059957945343282^16020202660644205002501507220006771421554169383555
98852914863182379 413490084170616
75093668333850551 **There is no limit!** 213796825837188
09183365675122131 2595567449219461
70238065059132456 982023131690176
78006675195485079921636419370285375124784014907159135459982790513399
61155179427110683113409058427288427979155484978295432^25245170652^2^226
9061394905987693002122963395687782878948440616007412 **Except for** 05
7164237715481632138063104590291613692670834285644073 **machine** 81
4657634732238502672530598997959960907994692017746248 **memory** 65
9250178329070473119433165550807568221846571746373296 74
57002440926616910874148385078411929804522981857338977648103126085903
00130241346718972667321649151113160292078173803343609024380470834040
3154190336

73

Big integers

C / C++
Fortran

int
INTEGER*4

long
INTEGER*8

long long
INTEGER*16

Out of the reach
of C or Fortran!

2

4

16

256

65536

4294967296

18446744073709551616

3402823669209384634…
63374607431768211456
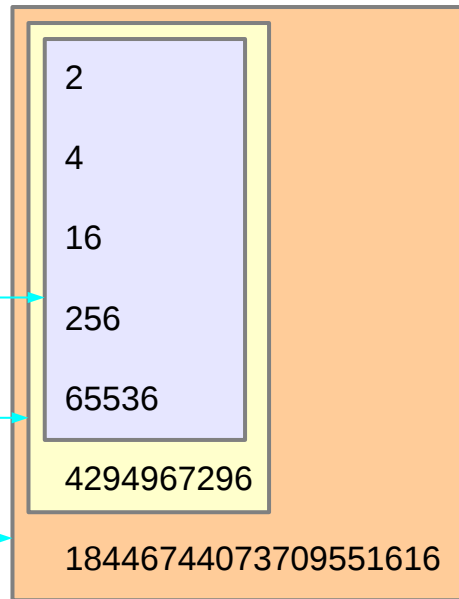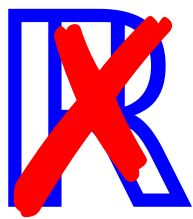
This may sound rather trivial but, in fact, Python is quite exceptional in this regard. The compiled languages have to allocate space for their integers in advance and so place limits on how large they can grow.

Floating point numbers

$\mathbb{R}$ ~~$\mathbb{R}$~~

1.0

0.33333333

3.14159265

2.71828182

And that's it for whole numbers. Now we will look at floating point numbers, a computer's way of storing fractional values. This is a computer's approximation to the real numbers. As we will see it is a problematic approximation.

The fancy $\mathbb{R}$ is the mathematical symbol for the real numbers, from the English word *Real*.

Basic operations

```
>>> 20.0 + 5.0
25.0

>>> 20.0 * 5.0
100.0

>>> 20.0 ** 5.0
3200000.0
```

```
>>> 20.0 - 5.0
15.0

>>> 20.0 / 5.0
4.0
```

Equivalent to integer arithmetic

76

For our basic operations, floating point numbers behave just the same as integers, using the same operators to achieve the same results. Floating point division creates a floating point number.

Floating point imprecision

```
>>> 1.0 / 3.0
0.3333333333333333
```

```
>>> 10.0 / 3.0
3.3333333333333335
```

If you are relying on this last decimal place, you are doing it wrong!

≈ **17** significant figures

So let's see our first problem.

Floating point arithmetic is not exact, and cannot be.

Floating point numbers on modern hardware tends to give a precision of 17 significant figures. You do see the occasional issue as shown on the slide but, frankly, if you are relying on the exact value of the final decimal place you are doing it wroing.

Hidden imprecision

```
>>> 0.1
0.1


>>> 0.1 + 0.1
0.2


>>> 0.1 + 0.1 + 0.1
0.3000000000000004
```

Really: if you are relying on this last decimal place, you are doing it wrong!

Not all imprecision is overt. Some of it can creep up on you.

Computers work in base 2. They can store numbers like ½ and ⅞ exactly. But they cannot store numbers like 1/10 exactly, just like we can't represent ⅓ exactly in a decimal expansion.

The errors in storing 1/10 are small enough, though, that they are invisible at first. However, if they accumulate they become large enough to show through.

Really: don't depend on precise values.

How big can a Python float be? — 1

```
>>> 65536.0**2
4294967296.0
```

So far, so good.

```
>>> 4294967296.0**2
1.8446744073709552e+19
```

Switch to "scientific notation"

1.8446744073709552 e+19

1.8446744073709552 ×10^{19}

79

Let's ask the same question about floats as we asked about integers: how large can they be?

We will repeat our approach of repeated squaring. We fast-forward to start at 65536.0 squared and notice that we soon get anomolous responses.

When we square 4,294,967,296 we get a number with the letter "e" in it. User's of pocket calculators at school may recognise this representation: it indicates a number between 1.0 and 9.999… multiplied by a power of 10.

Floating point numbers can only hold roughly 17 significant figures of accuracy. This means that when the integer needs more than 17 digits something has to give.

Floats are not exact

```
>>> 4294967296.0**2                    Floating point
1.8446744073709552e+19


>>> 4294967296**2                      Integer
18446744073709551616
```

$1.8446744073709552 \times 10^{19}$ ⟶ 18446744073709552000

－ 18446744073709551616
_____

384

The approximation isn't bad. The error is 384 in 18446744073709551616, or approximately 2×10^{-17}.

How big can a Python float be? — 2

```
>>> 1.8446744073709552e+19**2
3.402823669209385e+38

>>> 3.402823669209385e+38**2
1.157920892373162e+77

>>> 1.157920892373162e+77**2        So far, so good.
1.3407807929942597e+154

>>> 1.3407807929942597e+154**2      Too big!
OverflowError: (34,
'Numerical result out of range')
```

If we accept that our answers are now only approximate we can keep squaring. The "e-number" representation of scientific notation is accepted on input by Python.
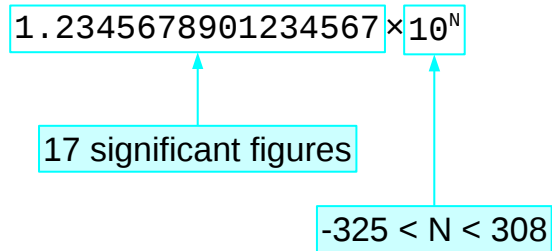
When we come to square $1.3407807929942597 \times 10^{154}$, though, we hit another issue, this one fatal.

We get an "overflow error". This means we have tried to create a floating point number larger than Python can cope with. Under some circumstances the "too big" problem gives rise to a sort-of-number called "`inf`" (standing for "infinity").

Floating point limits
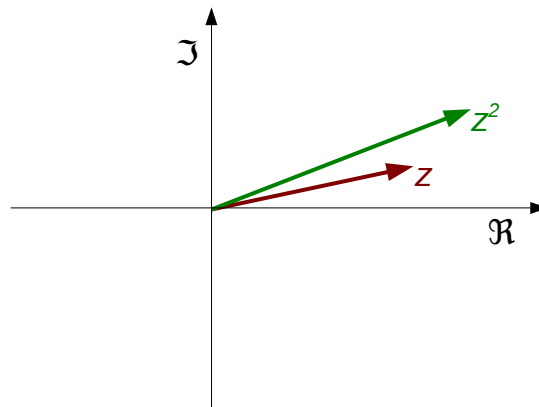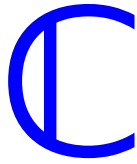
$$1.2345678901234567 \times 10^{N}$$

17 significant figures

$-325 < N < 308$

Positive values:

$4.94065645841 \times 10^{-324} < N < 8.98846567431 \times 10^{307}$

Just for the record, floating point numbers have limits both in terms of the largest and smallest numbers they can contain.

Complex numbers

$$\mathbb{C}$$

```
>>> (1.25+0.5j)**2
(1.3125+1.25j)
```

Python also supports complex numbers, using j for the square root of -1. We will not use them in this course, but you ought to know they exist.

Progress

Arithmetic	+ - * / ** %
Integers	No limits!
Floating point numbers	Limited size
	Limited precision
Complex numbers	

84

Exercise 4

Replace the XXXX in `exercise4.py` to evaluate
and print out the following calculations:

1. $223 \div 71$

2. $(1 + 1/10)^{10}$

3. $(1 + 1/100)^{100}$

4. $(1 + 1/1000)^{1000}$

3 minutes

85

Comparisons

5 < 10   ✔

5 > 10   ✘

We can do arithmetic on numbers. What else?
We need to be able to compare numbers.
Is 5 less than 10? Yes it is.
Is 5 greater than 10? No it isn't.

Comparisons

```
>>>  5 < 10          Asking the question
True                 ✔


>>>  5 > 10          Asking the question
False                ✘
```

Now let's see that in Python.

The "less than" character appears on the keyboard so we don't need anything special to express the concept like "**" for powers.

Python seems to answer the questions with "`True`" and "`False`".

True & False

```
>>>  type(True)
<class 'bool'>
```
"Booleans"

5 + 10 ⟶ 15   int

int   int

5 < 10 ⟶ True   bool

88

The important thing to understand is that this is not just Python reporting on a test but rather the value generated by the test. `True` and `False` are (the only) two values of a special Python type called a "Boolean" used for recording whether something is true or not.

Just as the "+" operator takes two integers and returns an integer value, the "<" operator takes two integers and returns a Boolean.

Booleans are named after George Boole, whose work laid the ground for modern algebraic logic. (His classic book's full title is "An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities", in true Victorian style.)

True & False

| bool | ✓ | True |
| bool | ✗ | False |
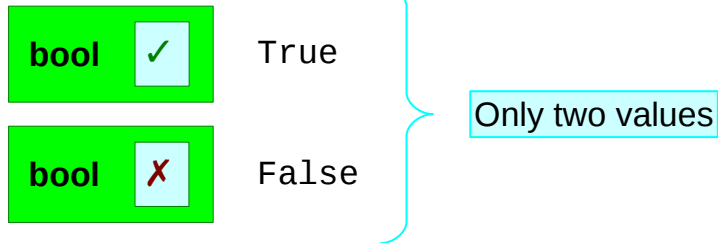
Only two values

The boolean type has precisely two values.

Six comparisons

Maths	Python	
=	==	Double equals sign
≠	!=	
<	<	
>	>	
≤	<=	
≥	>=	

There are six comparison operations in Python.

The equality comparison is defined in Python with a *double* equals sign, "==". The sign is doubled to distinguish comparison from assignment.

There is no "not equals" symbol on the standard keyboard. Instead, Python uses the "!=" pair of characters. (As with "**" there must be no space between the two characters.)

"Less than" and "greater than" we have already covered. These are implemented directly by the "<" and ">" characters.

There are no "less than or equal to" or "greater than or equal to" keys, though, so Python resorts to double character sequences again.

Equality comparison & assignment

=

```
name = value
```

Attach a name to a value.

==

```
value₁ == value₂
```

Compare two values

If ever there was a "classic typo" in programming it is the confusion of "=" and "==". Be careful.

Textual comparisons

```
>>> 'cat' < 'dog'          Alphabetic ordering

True


>>> 'Cat' < 'cat'          Uppercase before lowercase

True


>>> 'Dog' < 'cat'          All uppercase before lowercase

True
```

Booleans typically arise from comparisons. We can compare more than numbers (integers or floating point). We can also compare strings.

Text comparisons are based around the ordering of characters in the Unicode character set. Note that all the uppercase letters in the Latin alphabet precede all the lowercase letters. So any text that starts with an uppercase letter counts as "less than" any text that starts with a lowercase letter.

Ordering text is *complicated*

Python inequalities use Unicode character numbers.

This is over-simplistic for "real" use.

"Collation" is a whole field of computing in itself

Alphabetical order?

German: z < ö

Swedish: ö < z

Please note, however, that this is just a comparison of strings. It is not a general comparison of text. Ordering text is called "collation" and is a very compicated field.

For example, different languages order characters differently.

If you want to learn more about this field, start with the Unicode page on collation: `http://www.unicode.org/reports/tr10/`

"Syntactic sugar"

```
                          0 < number

0 < number < 10  ──────▶      and

                          number < 10



>>> number = 5

>>> 0 < number < 10

True
```

A common requirement is to determine if a number lies in a particular range. For this purpose, Python supports the mathematical notation $a < b < c$. The inequalities can be any combination that make sense.

Converting to booleans

`float()`   Converts to floating point numbers

                          `<class 'float'>`

`int()`   Converts to integers

                          `<class 'int'>`

`str()`   Converts to strings

                          `<class 'str'>`

`bool``()`   Converts to booleans

                          `<class 'bool'>`

95

As with all Python types there is a function named after the type that tries to convert arbitrary inputs into Booleans. Given that there are only two Boolean values this tends to be a very simple function.

Useful conversions

```
''  ─────────────► False          Empty string

'Fred' ─────────► True            Non-empty string



0  ─────────────► False           Zero

1  ─────────────► True            Non-zero

12 ─────────────► True

-1 ─────────────► True
```
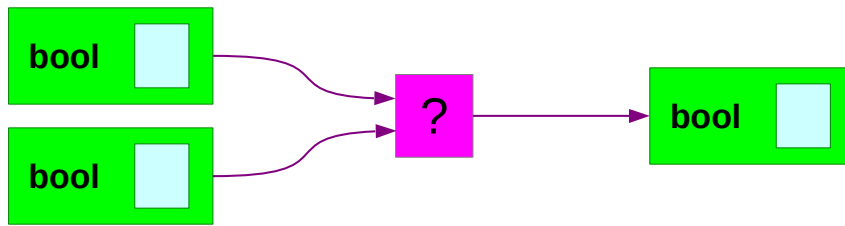
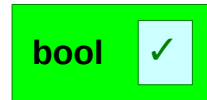The empty string is mapped to `False`. Every other string is mapped to `True`.

For integers, 0 is mapped to `False` and every other value to `True`.

For floating point numbers, 0.0 is mapped to `False` and every other value to `True`.

Boolean operations

bool [ ]
bool [ ]
→ **?** → **bool** [ ]
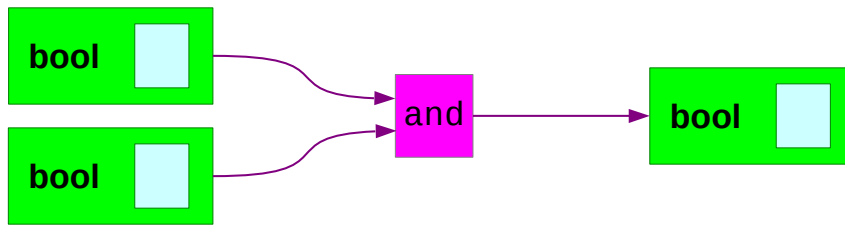
Numbers have +, –, * …

What do booleans have?
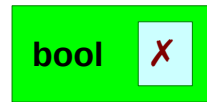
bool ✓

bool ✗

Boolean types have their own arithmetic just like ordinary numbers. It was the algebra of these that George Boole developed.

Boolean operations — "and"


```
True   and True  ──────────────▶  True        Both have
                                                to be True
True   and False ──────────────▶  False

False  and True  ──────────────▶  False

False  and False ──────────────▶  False
```

The first operation on Booleans is the "and" operator.

The and of two booleans values is `True` if (and only if) both its inputs are `True`. If either is `False` then its output is `False`.

```
>>> True and False
False
>>> True and True
True
```

Boolean operations — "and"

```
>>> 4 < 5 and 6 < 7        4 < 5 ──► True
                                          ⎫
                                          ⎬ and ──► True
True                       6 < 7 ──► True ⎭
```

```
>>> 4 < 5 and 6 > 7        4 < 5 ──► True
                                           ⎫
                                           ⎬ and ──► False
False                      6 > 7 ──► False ⎭
```

We are much more likely to encounter the input booleans as the results of comparisons that as literal values.

Boolean operations — "or"
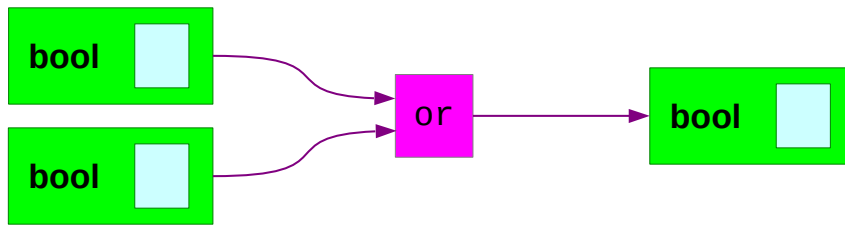

```
True   or   True  ───────────────►  True        At least
                                                 one has
True   or   False ──────────────►   True         to be True

False  or   True  ─────────────►    True

False  or   False ───────────────►  False
                                                     100
```

The next boolean operation to look at is "or". The results of this operation is
True if either of its inputs are True and False only if both its inputs are
False.

Boolean operations — "or"

```
>>> 4 < 5 or 6 < 7          4 < 5 ─► True
                                           ⎫
                                           ⎬ or ─► True
True                        6 < 7 ─► True  ⎭




>>> 4 < 5 or 6 > 7          4 < 5 ─► True
                                           ⎫
                                           ⎬ or ─► True
True                        6 > 7 ─► False ⎭
```

Again, we tend to encounter it more often with other tests than with literal booleans.

Boolean operations — "not"

bool [ ] ⟶ not ⟶ bool [ ]
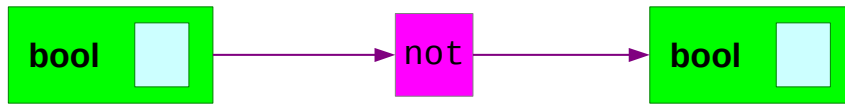
not True ⟶ False

not False ⟶ True

The final operation is "not". This takes only one input and "flips" it. True becomes False and *vice versa*.

Boolean operations — "not"

```
>>> not 6 < 7                6 < 7 ──► True ── not ──► False

False
```


```
>>> not 6 > 7                6 > 7 ──► False ── not ──► True

True
```

103

Ambiguity in expressions

$$3 + 6 / 3$$

✘   ✔

$(3 + 6) / 3$ $3 + (6 / 3)$

3   5
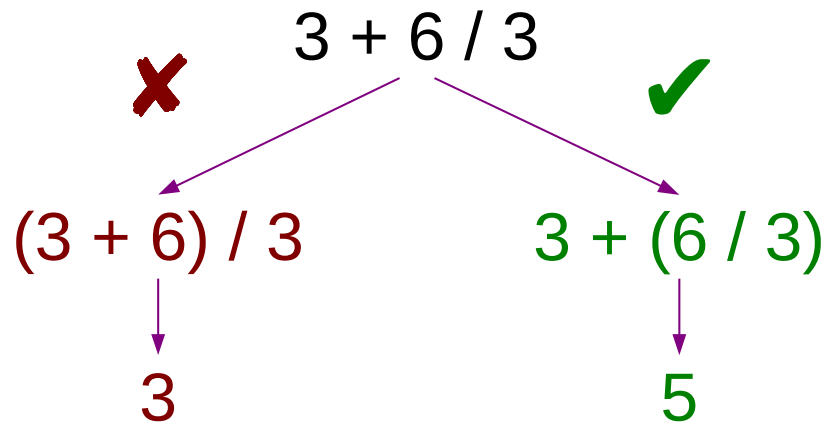
Division before addition

$$3 + 6 / 3$$

Division first

$$3 + \quad 2$$

Addition second

$$5$$

"Order of precedence"

First

x**y   -x   +x   x%y   x/y   x*y   x-y   x+y

x==y   x!=y   x>=y   x>y   x<=y   x<y

not x   x **and** y   x **or** y

Last

Progress

Comparisons	`== != < > <= >=`
Numerical comparison	`5 < 7`
Alphabetical ordering	`'dig' < 'dug'`
Booleans	`True    False`
Boolean operators	`and or not`
Conversions	`''` ⟶ `False`
	`0` ⟶ `False`
	`0.0` ⟶ `False`
	other ⟶ `True`

107

Exercise 5

1.   `'sparrow' > 'eagle'`

2.   `'dog' > 'Cat' or 45 % 3 == 0`

3.   `60 - 45 / 5 + 10 == 1`

3 minutes

108

Names and values: "assignment"

```
>>> alpha = 100
```

1. `alpha = ` **100**

    int `100`

    Python creates
    an "integer 100"
    in memory.

2. **alpha = ** `100`
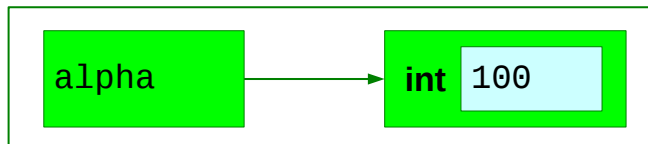
    Python attaches
    the name "alpha"
    to the value.

    alpha ⟶ **int** `100`

109

Now let's go back to the attaching of names to values that we saw with our `hello3.py` script.

Consider the simple Python instruction shown.

Python does two things, strictly in this order:

First, it notices the literal value 100 (an integer). So Python allocates a chunk of memory large enough and creates a Python object in it that contains a Python integer with value 100.

Second, it creates the name "`alpha`" and attaches it to the integer.

Assignment: right to left

`alpha = 100`

"RHS": right hand side
Evaluated first
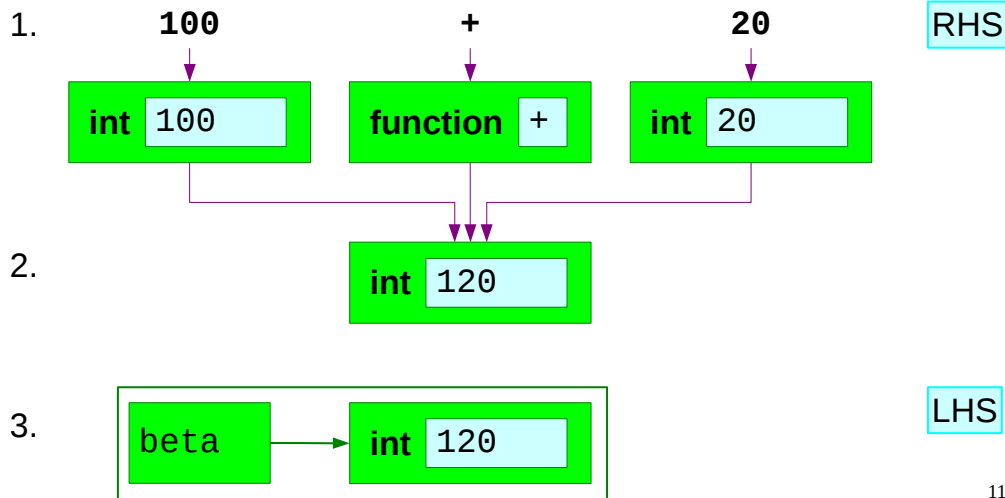
"LHS": left hand side
Processed second
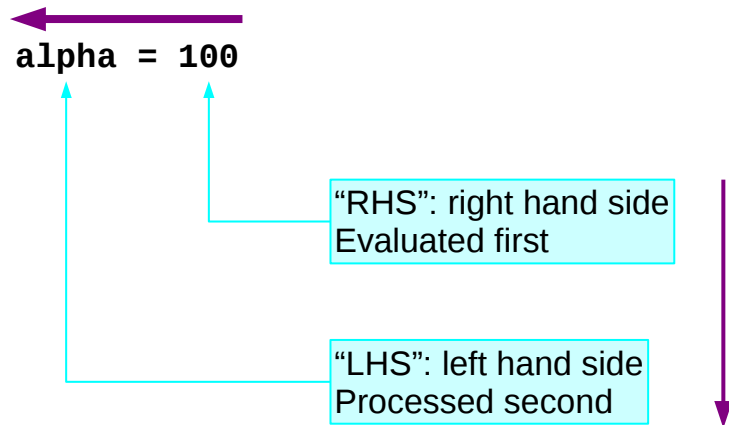
The key thing to note is that the processing happens right to left. Everything to the right hand side is processed first. Only after that processing is done is the left hand side considered.

In this example it's trivial. It will become less trivial very soon so remember that the right hand side is evaluated before the left hand side is even looked at.

ps: Computing uses the phrases "left hand side" and "right hand side" so often that they are typically written as "LHS" and "RHS".

Simple evaluations

```
>>> beta = 100 + 20
```

1. **100**   **+**   **20**   RHS

   int 100   **function** +   **int** 20

2.   **int** 120

3.   beta → **int** 120   LHS

111

We can see a slightly more involved example if we put some arithmetic on the RHS.

Again, the RHS is evaluated first.

First, Python notices three "tokens": the 100, the name "+" and the 20. It creates two integer objects just as it did with the previous example and it looks up a pre-existing function object that does addition of integers.

Second, Python triggers the addition function to generate a third integer with the value 120.

This completes the evaluation of the RHS.

Third, Python creates a name "beta" and attaches it to the freshly created integer 120.

Changing value — 1

```
>>> gamma = 1

>>> gamma = 2
```

Now let's look at attaching a name to a value and then changing that value. Suppose we have these two lines of Python.

In common with most—but not all—programming languages, Python lets you set a name for a value and then change it.
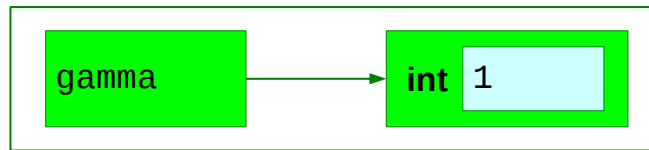
Changing value — 2

```
>>> gamma = 1
```

int 1

113

The process begins exactly as it did for the `alpha` example above.
The right hand side of the first line sets up an integer 1 in Python memory.

Changing value — 3

```
>>> gamma = 1
```

gamma → **int** 1

114

The left hand side of the assignment operation then attaches a name to that value, just as we saw before.

Changing value — 4

```
>>> gamma = 1

>>> gamma = 2
```

gamma → int 1

int 2
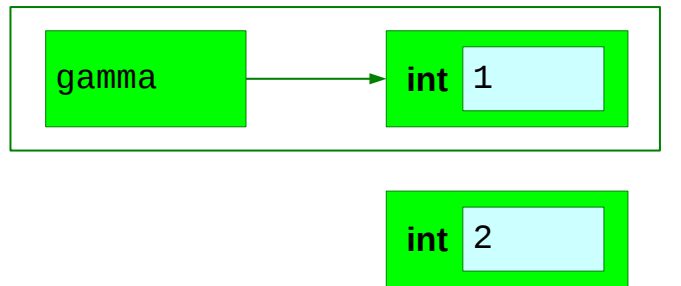
115

Now we move to the second line.

Again, Python processes the right hand side first. This is an integer 2 object and Python creates that in memory, alongside the existing integer 1 object.

That completes the processing of the second line's right hand side.

Changing value — 5

```
>>> gamma = 1

>>> gamma = 2
```

gamma

int 1

int 2
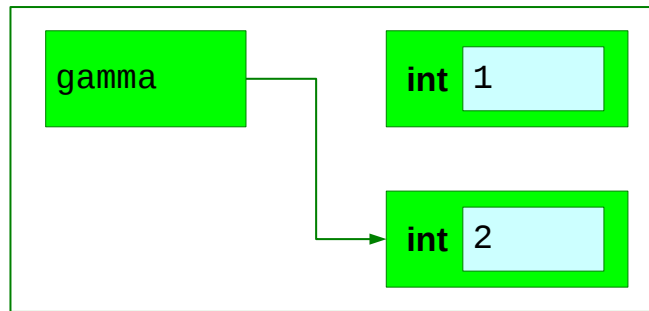
116

So Python then turns its attention to the second line's left hand side. This requires Python to attach the name gamma to the value just created. Given that the name already exists, Python detaches it from its old value and attaches it to the new one, leaving the old integer 1 object behind.

Changing value — 6

garbage collection

```
>>> gamma = 1

>>> gamma = 2
```
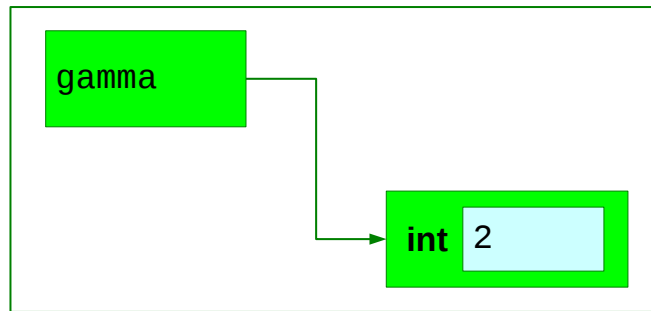
gamma → int 1 ✗

int 2
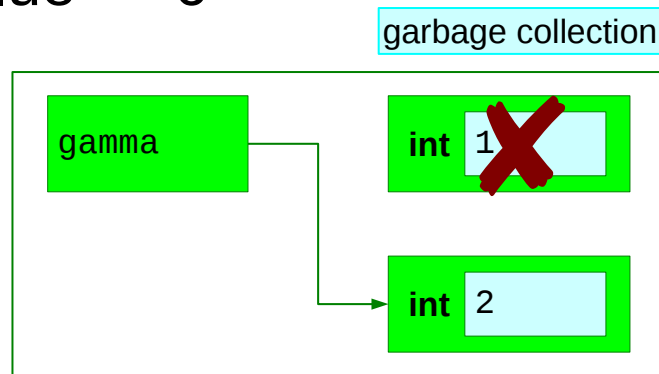
Now there are no remaining references to the old integer 1. Python automatically cleans it up, freeing the space for re-use.

This is a process called "garbage collection". In some languages you have to free up unused space yourself; in Python the system does it for you automatically.

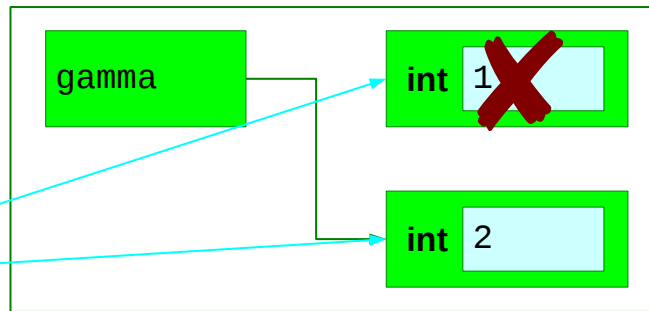# Changing value — 7

```
>>> gamma = 1

>>> gamma = 2
```

gamma

int 2

This leaves Python in its final state.

Changing value — a subtlety

```
>>> gamma = 1

>>> gamma = 2
```

gamma | int 1 ✗

int 2

Two *separate* integer objects.

Python doesn't change an integer's value;
Python *replaces* the integer.

Python integers are "**immutable**"

119

There is one last observation to make about this process.

Note that when we "changed gamma's value" what we did was to create a whole new integer object and reset gamma to point to it. We did not reach into the memory allocated to the integer object and change its value.

The fact that we cannot reach into Python integers and change their values makes Python integers "immutable".

Names on the RHS — 1
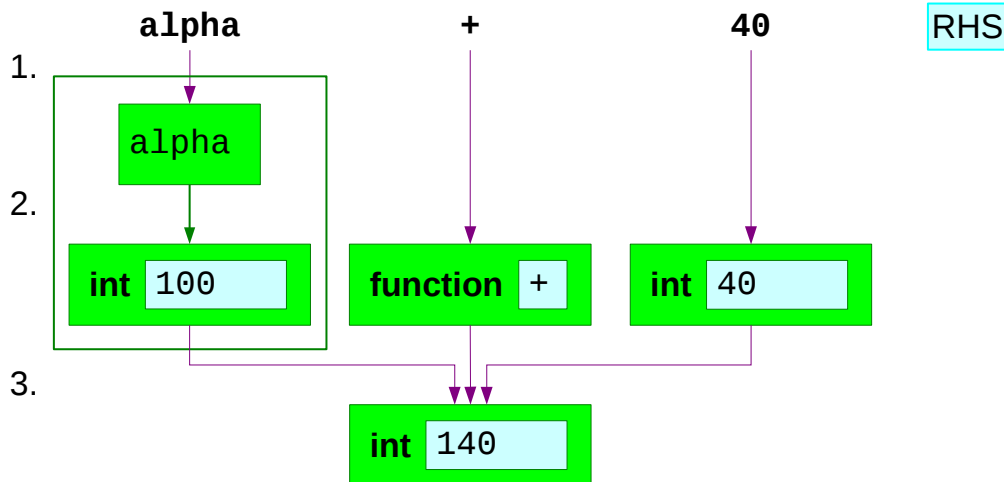
```
>>> delta = alpha + 40
```


Now we will consider a more significantly involved example, one with a name on the RHS.

First, Python recognizes the three tokens on the RHS. These are the name "alpha" the "+" and the literal integer 40.
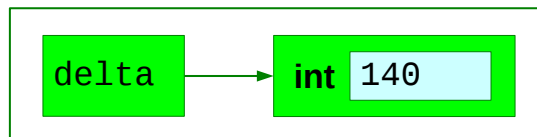
Second, it looks up the names. The "alpha" is replaced by the integer 100 and the name "+" is replaced by the actual function that does addition. The token 40 is replaced by the actual integer 40 in memory.

Third, it runs the function to give an integer 140 object in memory.

Names on the RHS — 2
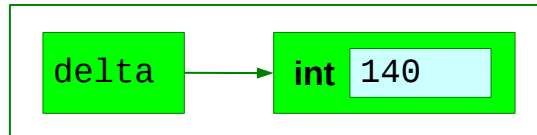
```
>>> delta = alpha + 40
```

4.

delta → int 140

LHS

Only after all that is the LHS considered, and the name "`delta`" is created and attached to the newly minted integer.

Same name on both sides — 0

delta → int 140
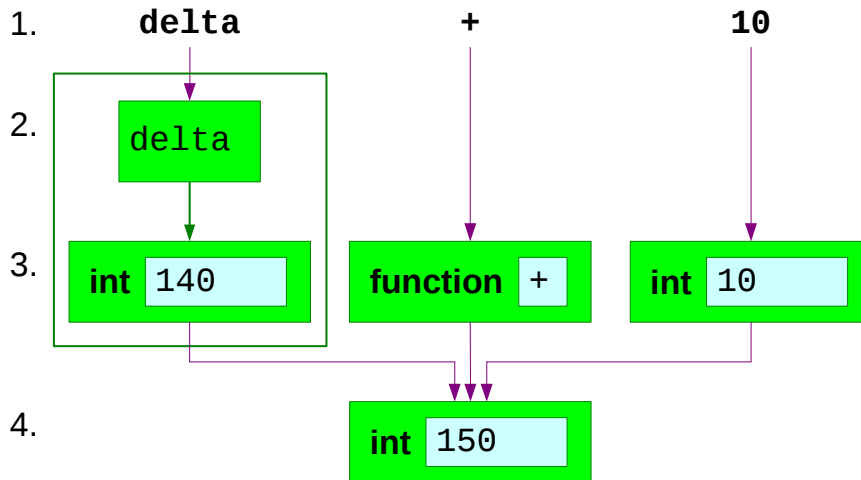
Starting position

```
>>> print(delta)

140
```

Now (finally!) we get to the interesting case.
We start with the name `delta` being attached to the value 140.

Same name on both sides — 1

```
>>> delta = delta + 10
```

1.   **delta**   **+**   **10**

2.   delta

3.   **int** `140`   **function** `+`   **int** `10`
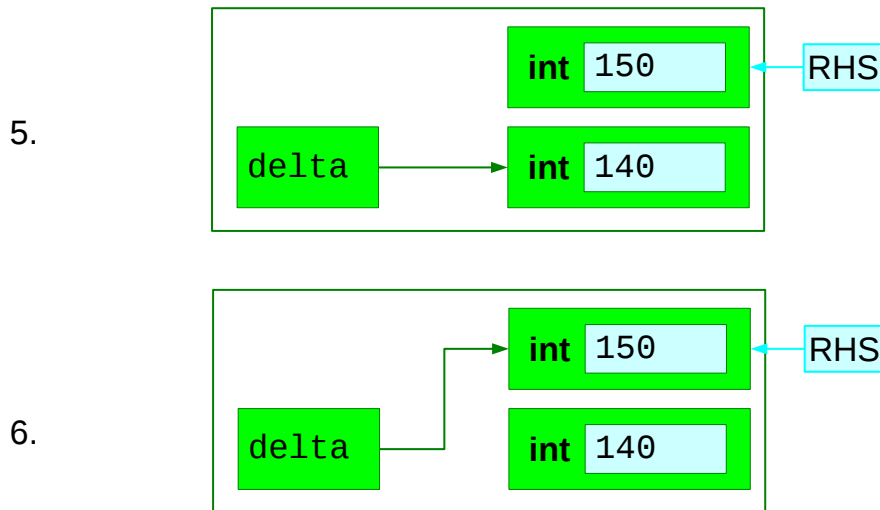
4.   **int** `150`

123

Then we run an assignment that has the name `delta` on *both* the left and right hand sides.

Again, first of all Python focuses exclusively on the RHS.

The expression "`delta + 10`" is evaluated to give rise to an integer 150 in Python memory.

Same name on both sides — 2

```
>>> delta = delta + 10
```

5.

| | int | 150 | ← RHS |
| delta → | int | 140 | |

6.

| | int | 150 | ← RHS |
| delta → | int | 140 | |

124

Only once that evaluation is complete does Python turn its attention to the LHS.

The name `delta` is going to be attached to the integer 150 in Python memory. No attention is paid to where the integer 150 came from.

The name `delta` is already in use and is attached to the integer 140. Its attachment is changed to the new integer 150.

Same name on both sides — 3

```
>>> delta = delta + 10
```

7.

```
int  150
delta    int  1̶5̶0̶
```

No longer used.

8.

```
delta    int  150
```
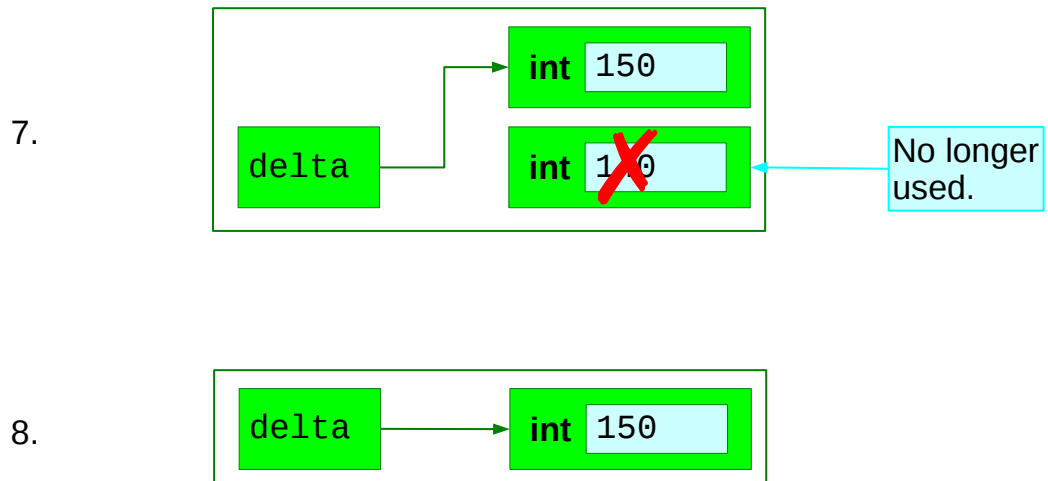
125

Again, the unreferenced, "left over", object is cleaned up by Python's automatic garbage collection.

"Syntactic sugar"

```
thing += 10                         thing = thing + 10

thing -= 10                         thing = thing - 10

thing *= 10                         thing = thing * 10
                    is equivalent to
thing /= 10                         thing = thing / 10

thing **= 10                        thing = thing ** 10

thing %= 10                         thing = thing % 10
```

The operation of modifying the value attached to a name is so common that Python, and some other languages, have short-cuts in their syntax to make the operations shorter to write. These operations are called "augmented assignments".

This sort of short-cut for an operation which could already be written in the language is sometimes called "syntactic sugar".

Deleting a name — 1

```
>>> print(thing)

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

Unknown variable

```
>>> thing = 1


>>> print(thing)

1
```

There's one last aspect of attaching names to values hat we need to consider. How do we delete the attachment?

First of all, let's see what it looks like when we refer to a name that isn't known to the system. The error message is quite straightforward:

```
name 'thing' is not defined
```

If we then create the name and attach it to a value, the integer 1 in this example, we can then use the name without error message.

Deleting a name — 2

```
>>> print(thing)

1
```
Known variable

```
>>> del thing


>>> print(thing)

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

To delete the attachment we use the Python command "`del`". The command

```
del thing
```

returns us to the state where the name is no longer known.

You can delete multiple names with the slightly extended syntax

```
del thing1, thing2, thing3
```

This is equivalent to

```
del thing1
del thing2
del thing3
```

but slightly shorter.

Progress

Assignment   `thing = thing + 10`

Deletion   `del thing`

Strictly right to left   `thing = thing + 10`

                                                 2nd   1st

+= etc. "syntactic sugar"   `thing += 10`

Our first "real" program

```
$ python3 sqrt.py
Number? 2.0
1.414213562373095
```

We have to write `sqrt.py`

First, the maths.
Then, the Python.

We now have enough to make a start on "real programming". We will need some more Python elements but we can meet them as we need them rather than up front.

We are going to write a program that prompts for a number and then calculates and prints out its square root.

First we will review the maths of calculating square roots so that we know what we are coding. Then we will do it in Python.

Square root of 2.0 by "bisection"
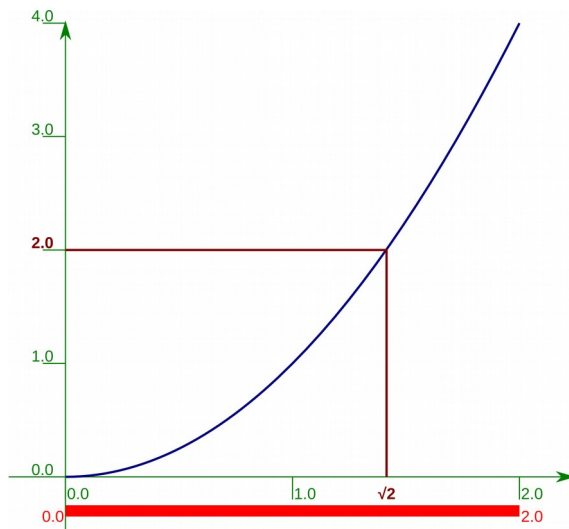
0.0 too small for √2

2.0 too large for √2

"Interval of uncertainty"

0.0 < √2 < 2.0

131

The technique we are going to use is called "bisection". If you know this technique you can relax for the next few slides. Please don't snore. ☺

We are going to go through it by hand for a few iterations because when we come to implement it in Python it is important that any confusion is due to the Python, and not the maths the Python is implementing.

The trick is to identify a range of values that must contain the actual value of √2. That is, we identify a lower bound that is less than √2 and an upper bound that is greater than √2.

We then have some way (which we will explain in the following slides) to improve that estimate by reducing the length of that interval of uncertainty. To be precise, we will cut the uncertainty in half which is why the process is called "bisection".

We start by taking a lower bound of $x=0$, which is definitely lower than $x=\sqrt{2}$ because $y=0^2=0<2$, and an upper bound of $x=2$, which is definitely higher than $x=\sqrt{2}$ because $y=2^2=4>2$.

Square root of 2.0 by bisection — 1
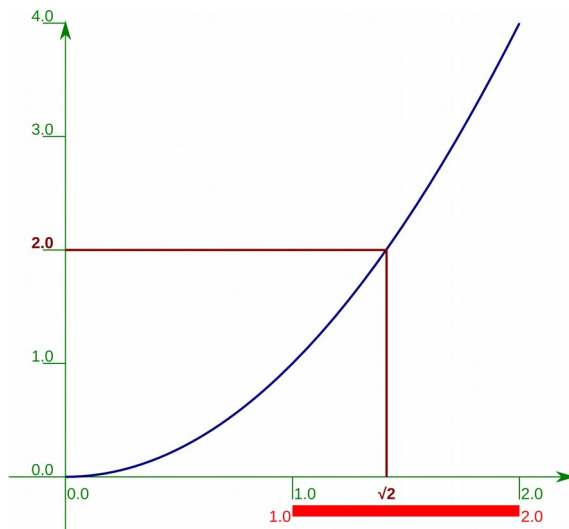

(0.0 + 2.0) / 2.0

1.0

1.0**2

1.0
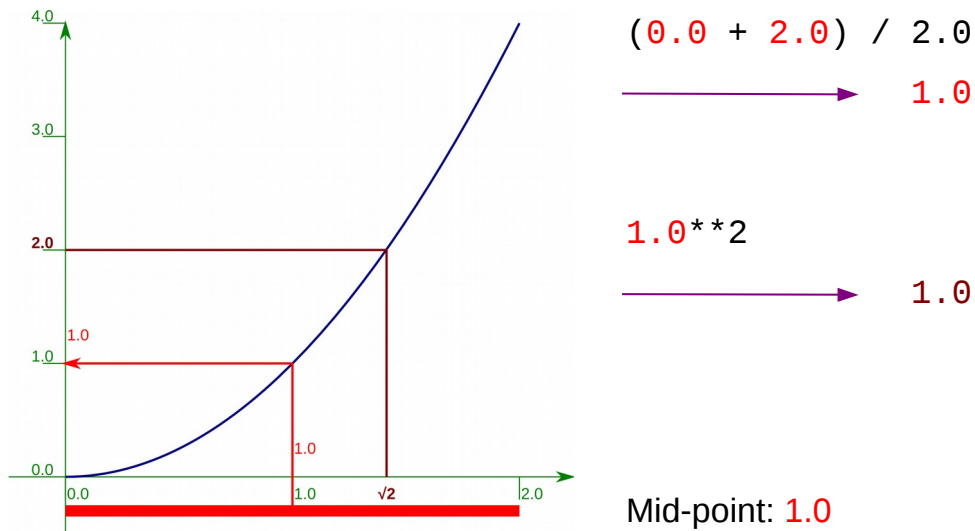
Mid-point: 1.0

So, what's the trick for halving the interval of uncertainty?

We find the midpoint of the interval. In this case it's obvious: the half-way point between x=0 and x=2 is x=1.

Then we square it to find its corresponding value of y. In this case $y=1^2=1$.

Square root of 2.0 by bisection — 2

$$\text{midpoint}^2 < \sqrt{2}^2$$

`1.0**2 < 2.0`

so change **lower** bound

$1.0 < \sqrt{2} < 2.0$

So what?

Well, *y*=1 is less than *y*=2 so the corresponding *x*-value, *x*=1 makes an acceptable lower bound for the interval of uncertainty. And if we change our lower bound to this value then our interval only runs from *x*=1 to *x*=2 with total length 1, rather than its original length 2.

We have halved our uncertainty.

If we can do this trick multiple times then we will reduce the interval of uncertainty very quickly to a length so small as to be irrelevant.

Square root of 2.0 by bisection — 3


```
(1.0 + 2.0) / 2.0
```

$\longrightarrow$ 1.5

```
1.5**2
```

$\longrightarrow$ 2.25

Mid-point: 1.5

134

So we do it again. The new mid-point lies at *x*=1.5. This has a corresponding *y*-value of *y*=2.25.

Square root of 2.0 by bisection — 4
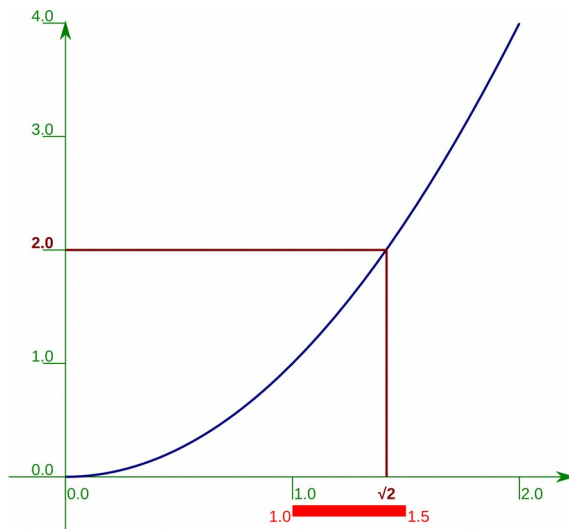
1.5**2 > 2.0

so change **upper** bound

1.0 < √2 < 1.5

y=2.25 is greater than *y*=2 so we can use the corresponding *x*-value of *x*=1.5 as our new upper bound. Now the interval of uncertainty is halved in length again to be ½.

Square root of 2.0 by bisection — 5

(1.0 + 1.5) / 2.0

$\longrightarrow$ 1.25

1.25**2

$\longrightarrow$ 1.5625

Mid-point: 1.25

We find the new mid-point again, x=1.25. Squaring this gives the corresponding y-value y=1.5625.
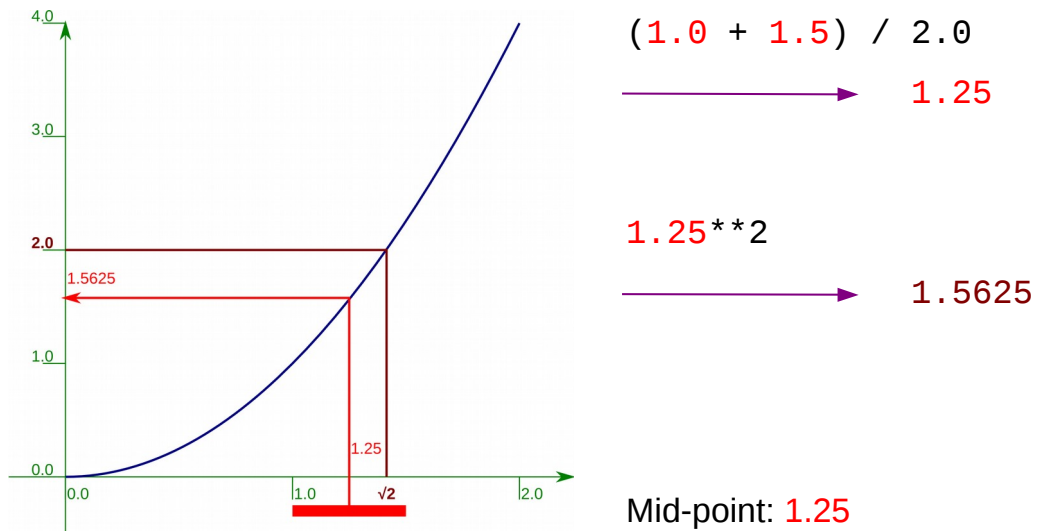
Square root of 2.0 by bisection — 6
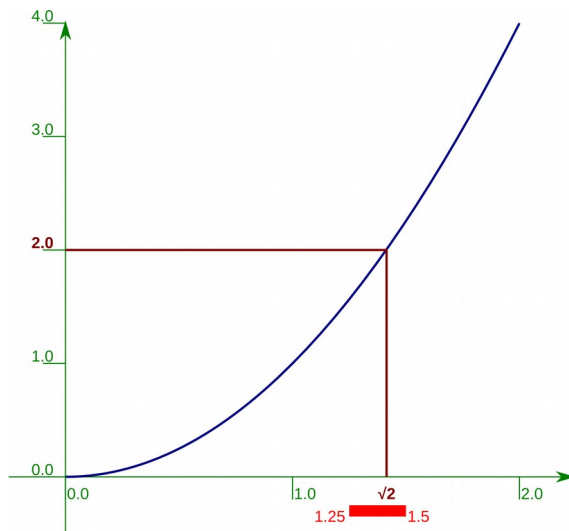
1.25**2 < 2.0

so change **lower** bound

1.25 < √2 < 1.5

137

y=1.5625 is less than y=2 so we change the lower bound. Our interval of uncertainty now has length ¼.

Square root of 2.0 by bisection — 7

(1.25 + 1.5) / 2.0

1.375

1.375**2

1.890625

Mid-point: 1.375

138

And again…

Square root of 2.0 by bisection — 8
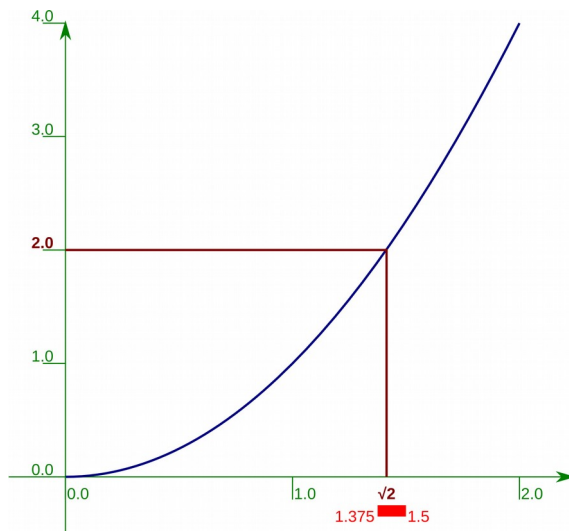
`1.375**2 < 2.0`

so change **lower** bound

$1.375 < \sqrt{2} < 1.5$

139

…to give an interval of length ⅛.

Exercise 6
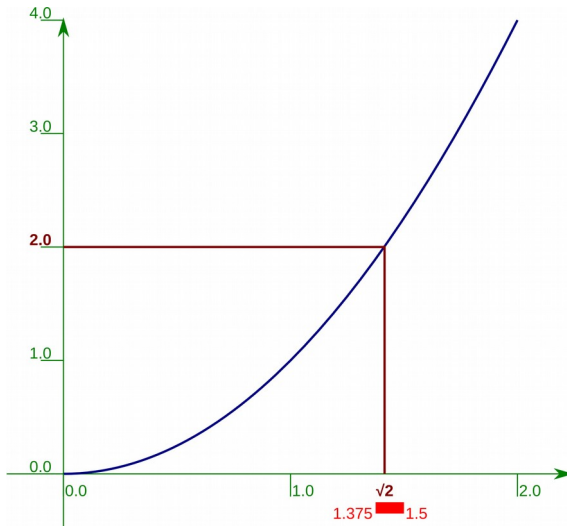
One more iteration.

Find the mid-point.
Square it.
Compare the square to 2.0.

**Do you change the
lower or upper bound?**

2 minutes

1.375 < √2 < 1.5

Please make sure that you understand the principle and do one more
iteration by hand.

Understanding before Programming

Apologies for spending so long on the maths but this is a general situation.
You must understand the situation before you can program it.

And now using Python!

```
lower = 0.0
upper = 2.0


middle = (lower+upper)/2
```

So now let's start the implementation of this process in Python.

We will do exactly the same maths, but this time with Python syntax.
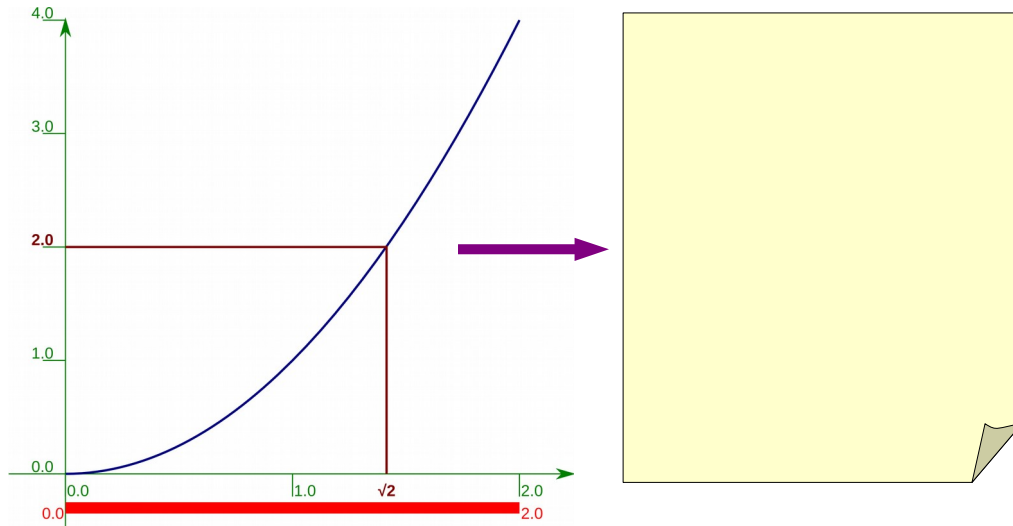
First we set the end points for the interval of uncertainty and attach names to the two *x*-values.

The names `lower` and `upper` are attached to the end points:

```
lower = 0.0
upper = 2.0
```

We establish the *x*-value of the mid-point and attach the name `middle` to it:

```
middle = (lower+upper)/2
```

This is exactly where we started last time, but we have attached Python names to the values.

And now using Python — 2


```
middle**2 < 2.0

          ↓

True


lower = middle

print(lower, upper)

       1.0     2.0
```

Next, we find the *y*-value corresponding to the mid-point (by squaring the *x*-value 1.0) and ask if it is less than 2.0, the number whose square root we are looking for.
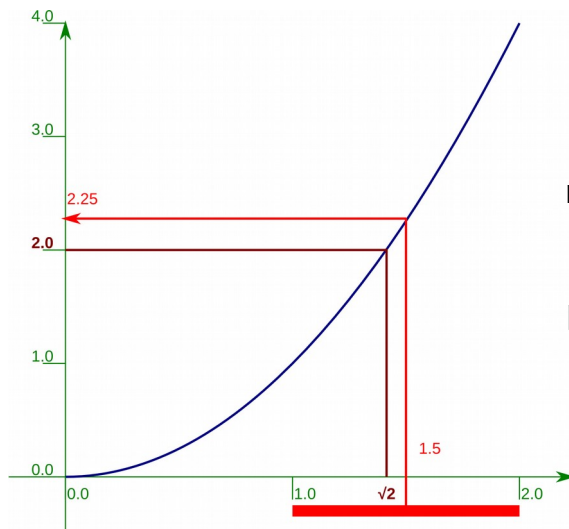
```
middle**2 < 2.0
```

Recall that this will return a Python boolean value: `True` or `False`.

The squared value is 1.0 which is less than 2.0 (i.e. we get `True`) so we raise the lower limit of the interval to the mid-point.

```
lower = middle
```

In this example we print the *x*-value at each end of the interval to track our progress.

And now using Python — 3


```
middle = (lower+upper)/2

print(middle, middle**2)
```
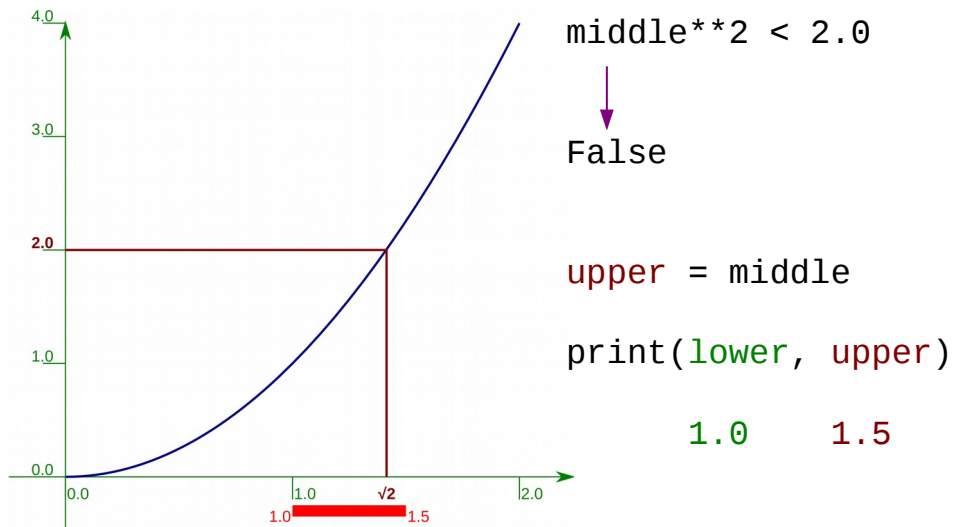
1.5   2.25

So we do it again.

We re-calculate the *x*-value for the mid-point. Note that because we changed the value the name lower was attached to the Python instruction is identical to the one we gave first time round:

```
middle = (lower+upper)/2
```

We do some additional printing to track progress.

And now using Python — 4


```
middle**2 < 2.0
```

False

```
upper = middle
```

```
print(lower, upper)
```

```
      1.0    1.5
```

Again, we ask if the mid-point's *y*-value (i.e. its *x*-value squared) is above or below our target of 2.0:

```
middle**2 < 2.0
```

and this time get a boolean `False`. Because the value is greater than 2.0 (our test evaluates to `False`) we change the value of the upper bound of the interval by attaching the name upper to the *x*-value of the mid-point:

```
upper = middle
```

The values being handled are exactly the same as they were when we did it as "raw maths" but this time they have names.

And now using Python — 5
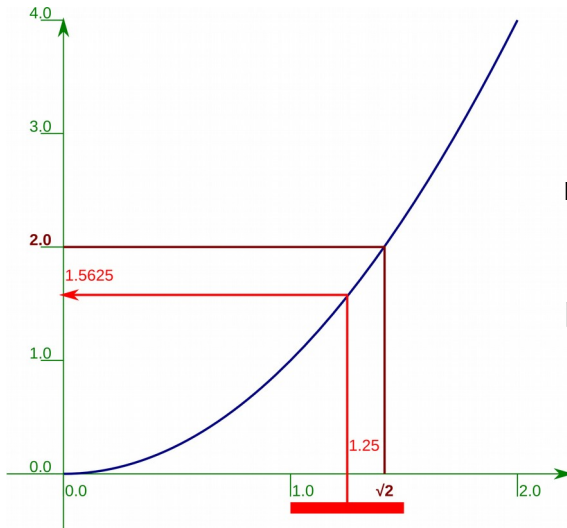

```
middle = (lower+upper)/2

print(middle, middle**2)
```

1.25   1.5625

We now do a third iteration.

And now using Python — 6


```
middle**2 < 2.0
```

True

```
lower = middle

print(lower, upper)
      1.25    1.5
```
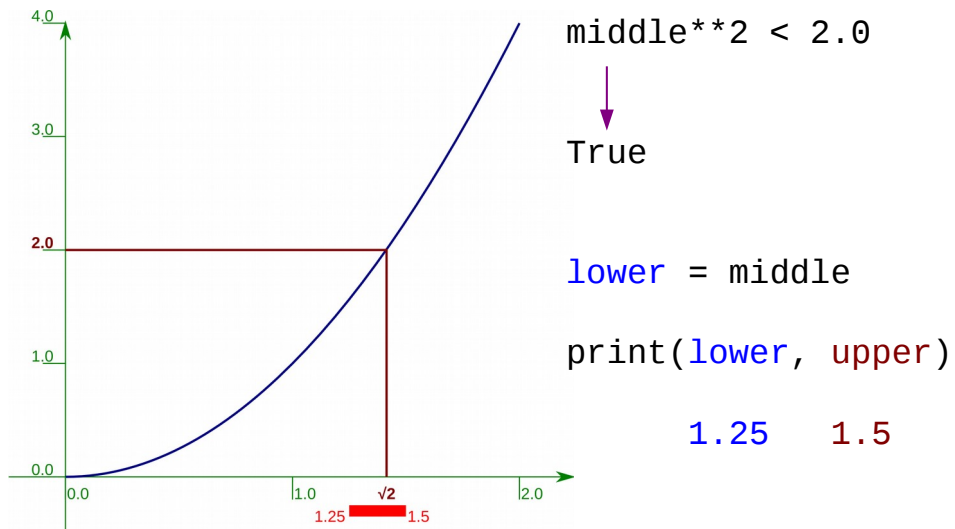
This time the test evaluates to `True` so we change `lower`.

And now using Python — 7
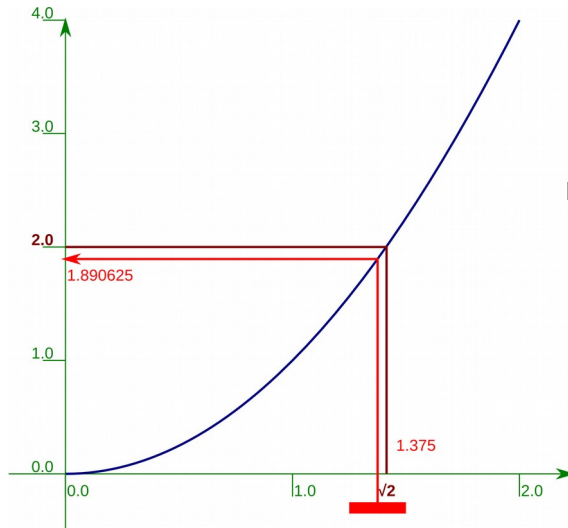

```
middle = (lower+upper)/2

print(middle, middle**2)
```

1.375   1.890625

Fourth iteration.

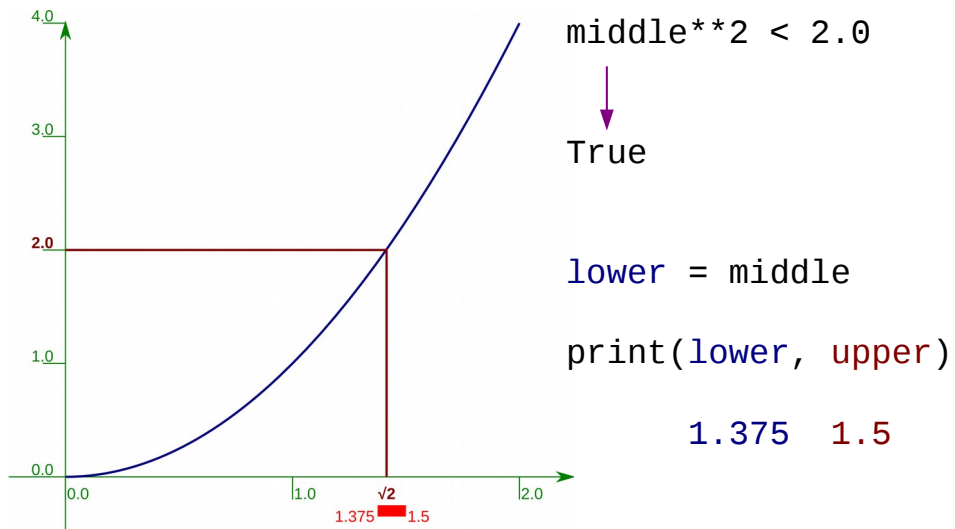# And now using Python — 8


```
middle**2 < 2.0
```

↓

```
True


lower = middle

print(lower, upper)

      1.375  1.5
```

And another `True` so we change `lower` again.
And that's enough of stepping through it manually.

Looking at the Python code

```
lower = 0.0
upper = 2.0

middle = (lower+upper)/2
print(middle, middle**2)

middle**2 < 2.0

✔              ?              ✘
lower = middle        upper = middle

print(lower, upper)
```
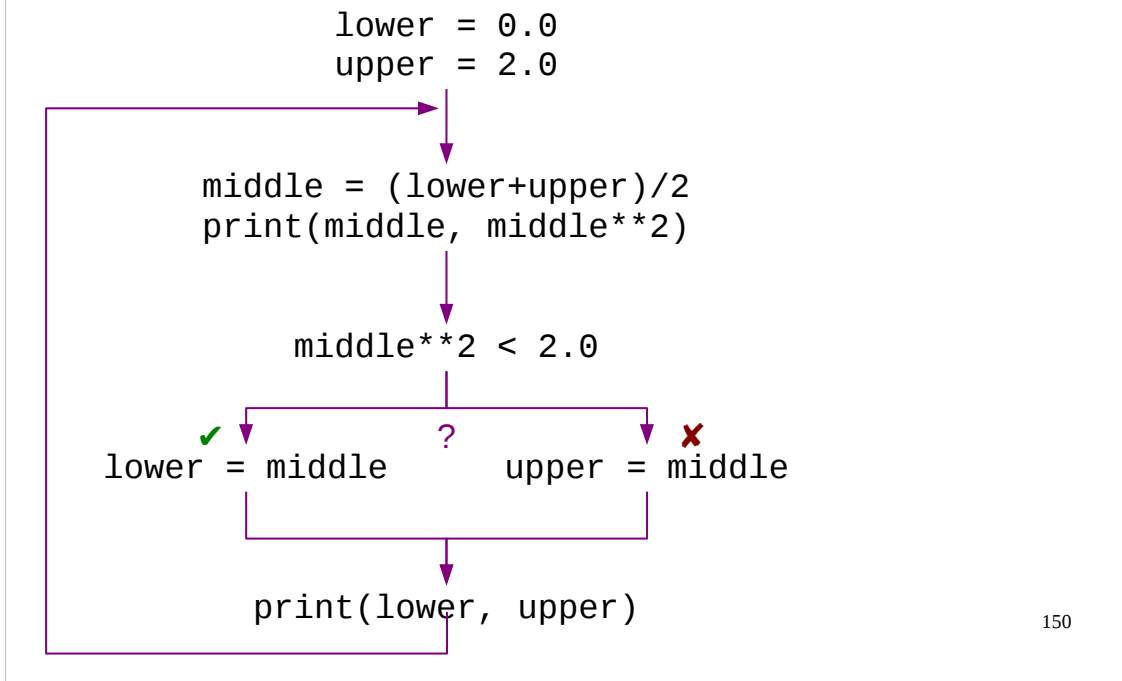
Let's look at the Python code we have used.

We started by initializing our interval of uncertainty:

　　　lower = 0.0

　　　upper = 2.0

Then we started the operations we would repeat by calculating the *x*-value of the mid-point:

```
middle = (lower+upper)/2
```

We squared this and compared the squared *y*-value with 2.0, our target value:

```
middle**2 < 2.0
```

and, based on whether this evaluated to True or False we ran either:

```
lower = middle
```

or:

```
upper = middle
```

Then we ran the iteration again.

Looking at the Python structures

```
lower = 0.0
upper = 2.0
```

```
middle = (lower+upper)/2
print(middle, middle**2)
```

```
middle**2 < 2.0
```

✔   ?   ✘

```
lower = middle        upper = middle
```

```
print(lower, upper)
```

151
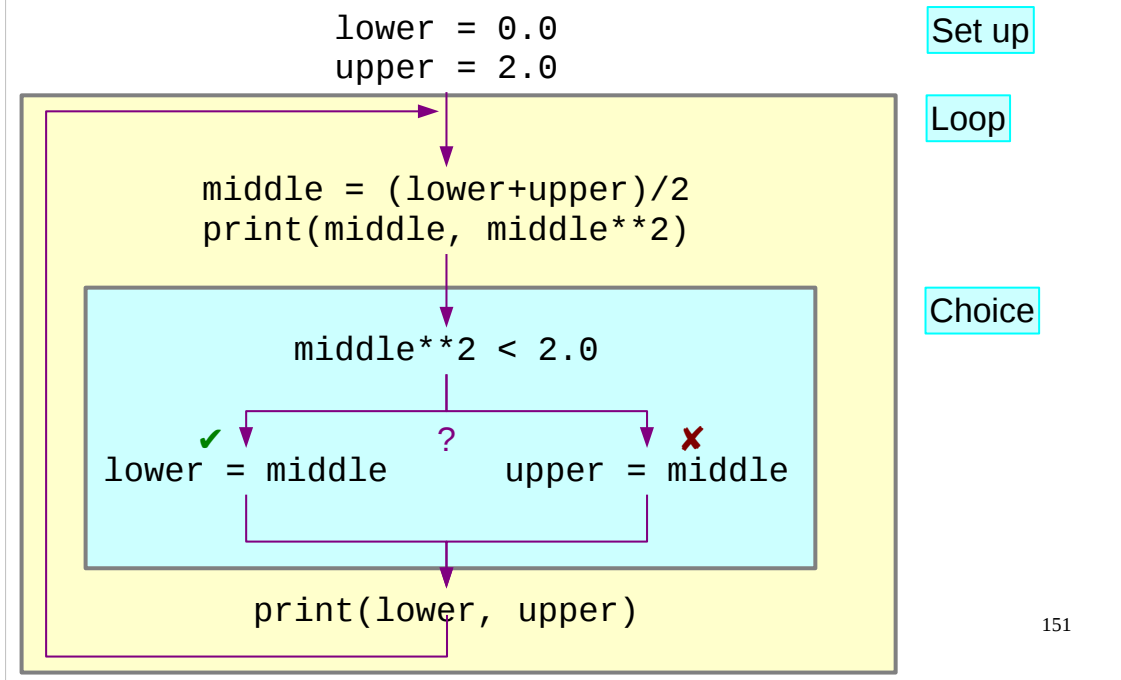
Structurally, we need to be able to do two things beyond our current knowledge of Python. We need to be able to run certain instructions time and time again ("looping") and we need to be able to choose one of two different actions depending on whether a boolean value is `True` or `False`.

Looping

Before

Loop test

✘   ✔

Loop body

After

Should the loop run (again)?

What is run each loop?
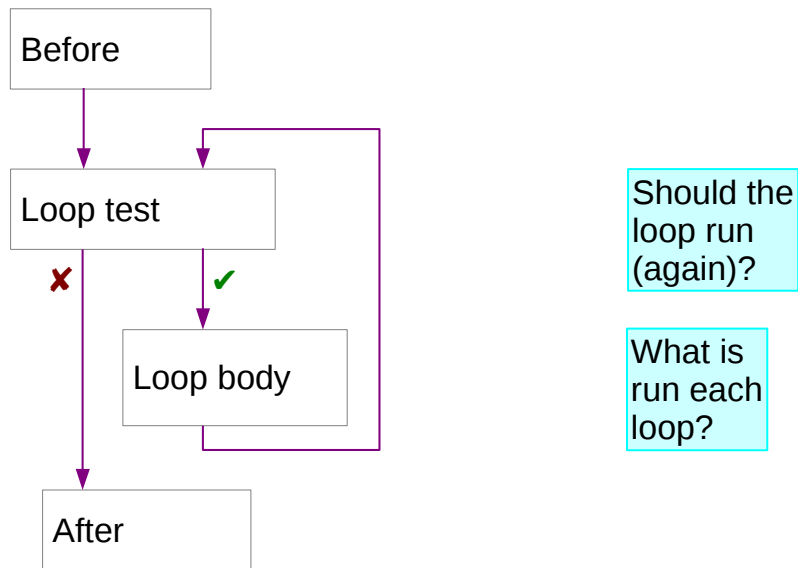
We will address looping first.

A loop has a number of components.

Strictly not part of the loop are the "before" and "after" sections but these give context and may use values needed by the loop.

The loop itself must have some sort of test to indicate whether the loop should run again or whether the looping can stop and control can pass to the "after" code.

Then there must be the set of instructions that will be run each time the loop repeats. We call this the "body" of the loop.

Loop example: Count from 1 to 10

Before	`number = 1`
Loop test	`number <= 10`
Loop body	`print(number)` `number += 1`
After	`print('Done!')`
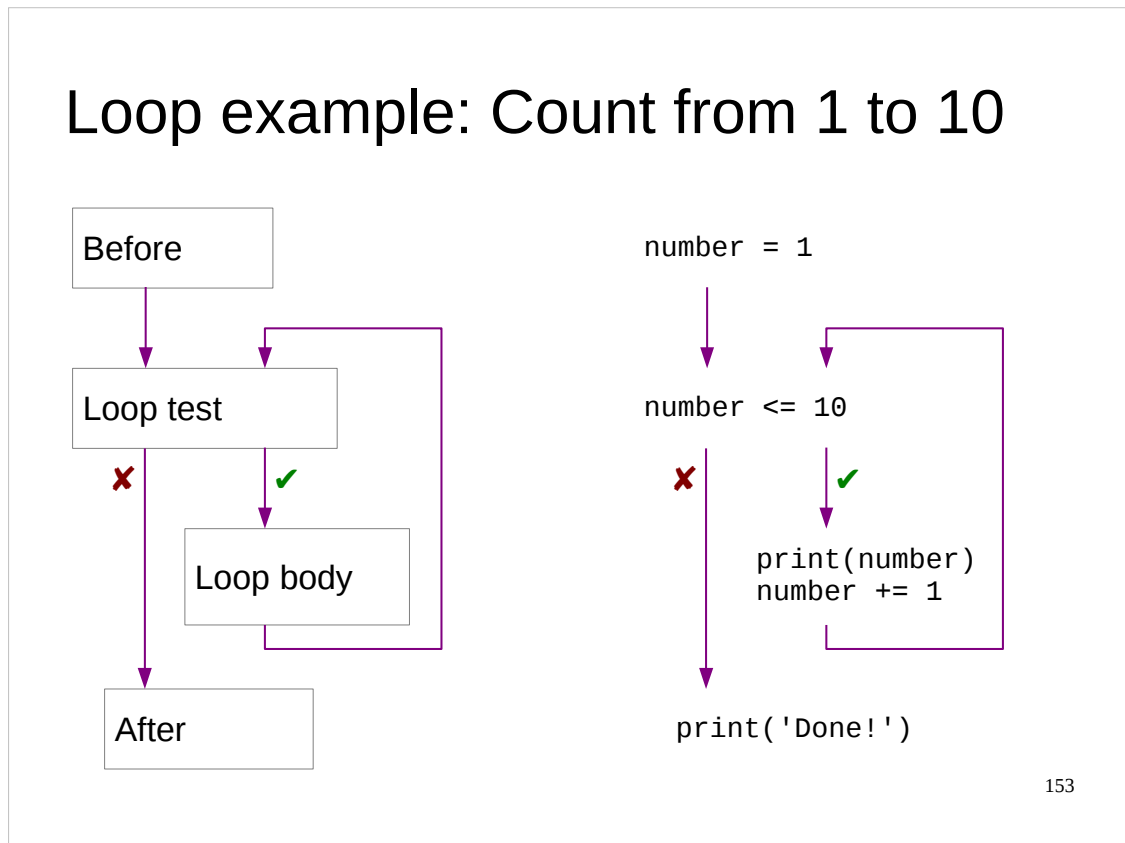
Let's consider an example that's simpler than our square root loop: counting from 1 to 10.

Our "before" block initializes the attachment of a name `number` to a value `1`:

```
number = 1
```

Our test sees if `number` is attached to a value less than or equal to `10` (our final value):

```
number <= 10
```

Recall that this evaluates to a boolean value.

If the test evaluates to `True` then we run the loop body. This has two lines, the first to print the value of `number` and the second to increase it by one:

```
print(number)
number += 1
```

If the test evaluates to `False` then we don't loop and exit the structure. We have a pointless `print` statement as a place-holder for more substantive code in serious scripts:

```
print('Done!')
```

This is what we want to encode in Python.

Loop example: Count from 1 to 10

```
number = 1

while number <= 10 :

␣␣␣␣ print(number)
␣␣␣␣ number += 1

print('Done!')
```

```
number = 1

number <= 10
        ✘        ✔

        print(number)
        number += 1

print('Done!')
```

154

This is how we encode the structure in Python. We will examine it element by element, but at first glance we observe a "`while`" keyword and a colon on either wise of the test and the loop body being indented four spaces.

Loop test: Count from 1 to 10

```
number = 1


while number <= 10 :



     print(number)
     number += 1


print('Done!')
```

"while" keyword

loop test

colon

We will start by looking at what we have done to the test. The test itself is
```
    number <= 1
```
which is a Python expression that evaluates to a boolean, `True` or `False`.

We precede the test expression with the Python keyword "`while`". This is what tells Python that there's a loop coming. It must be directly followed by an expression that evaluates to a Boolean.

We follow the test expression with a colon. This is the marker that the expression is over and must be the last element on the line.
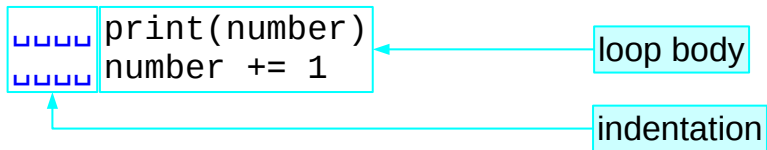
Note that the test evaluates to True for the loop to be run and False for the loop to quit. We are testing for "shall the loop keep going" not "shall the loop stop". Python tests for *while*, not *until*.

Loop body: Count from 1 to 10

```
number = 1


while number <= 10 :


    print(number)
    number += 1

print('Done!')
```

loop body

indentation

The loop body, the code that is repeated, appears on the lines following the "`while` line". Both its lines are indented by four spaces each.

Note that the "after" section is not indented.

Loop example: Count from 1 to 10

```
number = 1

while number <= 10 :

    print(number)
    number += 1


print('Done!')
```

          while1.py

```
$ python3 while1.py
1
2
3
4
5
6
7
8
9
10
Done!
$
```

First let's check that this really works. In the file `while1.py` in your home directories you will find exactly the code shown in the slide. Run it and watch Python count from 1 to 10.

Python's use of indentation

```
number = 1

while number <= 10 :
```
Four spaces' indentation indicate a "block" of code.

```
    print(number)
    number += 1
```
The block forms the repeated lines.

```
print('Done!')
```
The first unindented line marks the end of the block.

158

The four spaces of indentation are not cosmetic. A sequence of lines that have the same indentation form blocks in Python and can be thought of as a single unit of code. In this case both lines get run (again) or neither of them do.

The indented block is ended by the first line that follows it with no indentation.
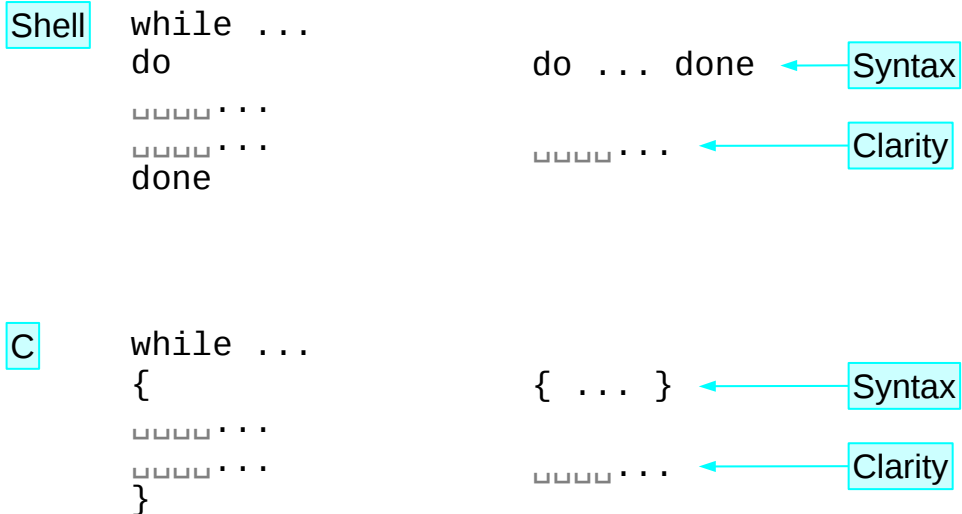
c.f. "legalese"

CHAPTE

BOARDS AND S

1. There shall be in the University
  (*a*) such Boards and Syndicates as may by ar
       maintained;
  (*b*) the following Boards and Syndicates, the cor
    (i) the Board of Graduate Studies, which sh
        of students as Graduate Students and th
        respect of graduate study or contributio
        assigned to it by Ordinance;
    (ii) the Board of Examinations, which shall ex
         of University examinations and other co
    (iii) the Local Examinations Syndicate, w
          examinations in schools and other instit
  (*c*) any other Boards or Syndicates the compos
       the University.
2. Any Board or Syndicate constituted by Statute
shall have the right of reporting to the University.
3. No person shall be appointed or reappointed a
or Managers even though it be not expressly calle
occasional Syndicate, who at the commencement of
service, as the case may be, would have attained the

1
1(a)
1(b)
1(b)(i)
1(b)(ii)
1(b)(iii)
1(c)
2
3

159
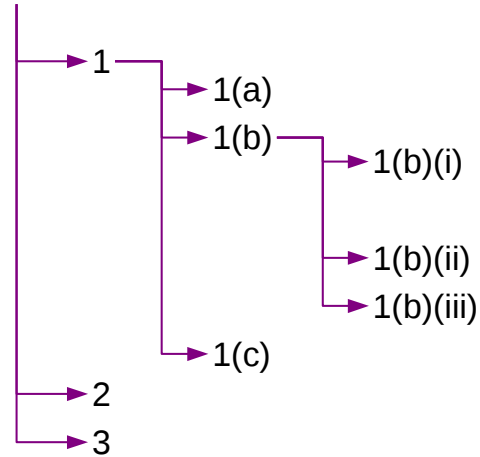
If this seems a little alien consider the "legalese" of complex documents. They have paragraphs, sub-paragraphs and sub-sub-paragraphs etc., each indented relative to the one containing them.

Other languages

```
Shell    while ...
         do                     do ... done  ←── Syntax
             ...
             ...                    ...      ←── Clarity
         done



C        while ...
         {                      { ... }      ←── Syntax
             ...
             ...                    ...      ←── Clarity
         }
```

Marking blocks of code is one of the places where computing languages differ from one another.

Some have special words that appear at the start and end of blocks like "do" and "done". Others use various forms of brackets like "{" and "}".

Interestingly, programmers in these languages typically also indent code within the blocks for visual clarity. Python simply uses the indentation for its core syntax rather than just for ease of reading.

Purely for interest, the Shell and C versions of `while1.py` are also in your home directory as `while1.sh` and `while1.c`.

Progress

while ... :

test to keep looping

code blocks

␣␣␣␣indentation

while *test* :
␣␣␣␣*action*$_1$
␣␣␣␣*action*$_2$
␣␣␣␣*action*$_3$

161

Exercise 7

For each script:

    ☐☐☐☐ Predict what it will do.

    ☐☐☐☐ Run the script.

    ☐☐☐☐ Were you right?

`while2.py`

`while3.py`

`while4.py`

`while5.py`

`while6.py`

To kill a running script: Ctrl + C

5 minutes

162

Back to our square root example

uncertainty

tolerance

0.0 ████████████ 2.0

2.0

$\times\frac{1}{2}$

1.0 ████ 2.0

1.0

$\times\frac{1}{2}$

1.0 ██ 1.5

0.5

$\times\frac{1}{2}$

1.25 ██ 1.5

0.25

$\times\frac{1}{2}$

1.375 █ 1.5

0.125

1.0×10^{-15}

What we want

What we get
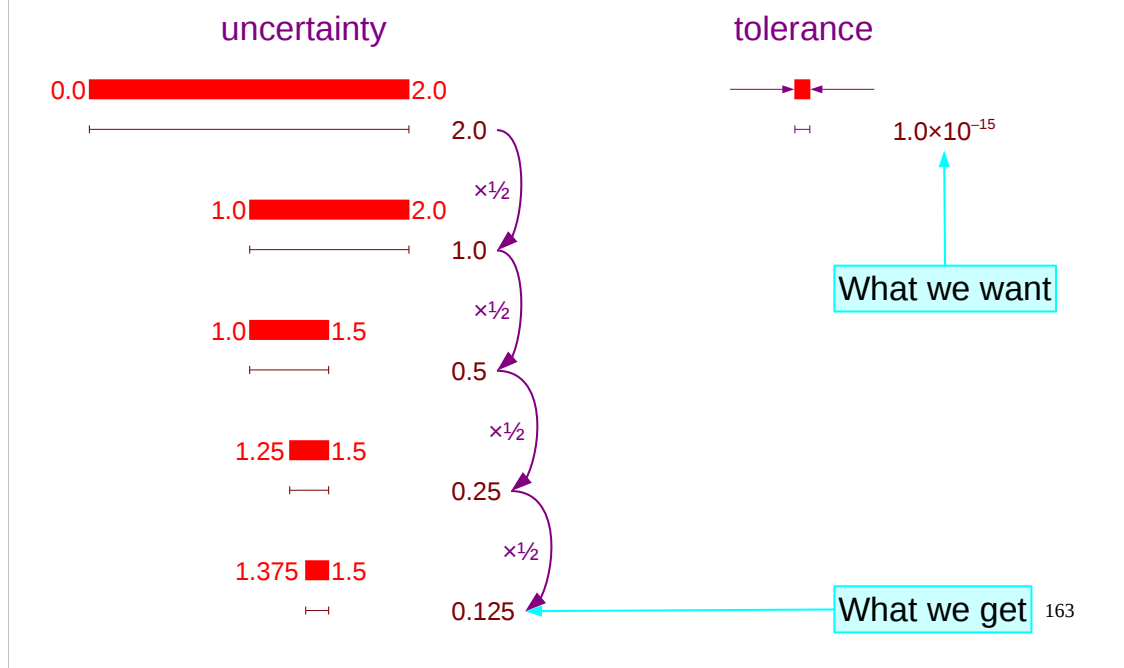
Now let's return to our square root example. We have a loop the body of which halves the length of the interval of uncertainty. We need to put this into a Python loop so we need a corresponding loop test.

One typical approach is to test to see if the interval is longer than some acceptable value. In this case we will demand that the interval have length no longer than 10^{-15}. (It will take 51 halvings to get from an initial length of 2.0 to something less than 10^{-15}.)

A common name for an "acceptable uncertainty" is a "tolerance": the amount of uncertainty we are prepared to tolerate.

Keep looping while … ?

uncertainty   >   tolerance

```
while uncertainty > tolerance :
```
⊔⊔⊔⊔ Do stuff.
⊔⊔⊔⊔
⊔⊔⊔⊔
⊔⊔⊔⊔

164

We need a Python test for this. Recall that Python needs a test that evaluates to `True` for the loop body to run. Our test then is "is the current uncertainty larger than the acceptable tolerance?"

We will set a name, `tolerance`, to have the value `1.0e-15`, calculate an uncertainty each loop and perform the test

```
uncertainty > tolerance
```

If this is `True` then we need to keep going.

If it is `False` then we can stop.

Square root: the loop

```
tolerance = 1.0e-15
lower = 0.0
upper = 2.0
uncertainty = upper - lower

while uncertainty > tolerance :

    middle = (lower + upper)/2

                ?

    print(lower, upper)
    uncertainty = upper - lower
```

Set up

Loop

Choice

165

So, if we return to our basic structure we can now see how Python's `while` syntax fits in.

We establish a tolerance.

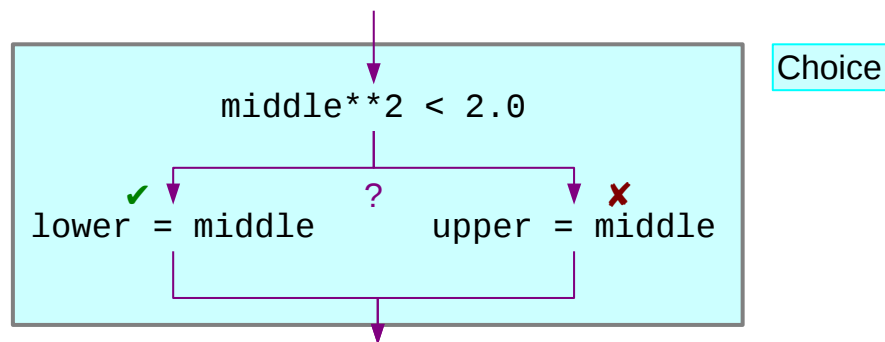We establish an initial uncertainty.

We test for

```
uncertainty > tolerance
```

as the loop test.

We recalculate the uncertainty at the end of the loop block for use in the next round of the test.

All we have to do now is to add in the choice block.

Choosing

```
                                                    Choice
       middle**2 < 2.0

   ✔                   ?              ✘
 lower = middle          upper = middle
```

```
middle**2 < 2.0 ————————▶ True or False

True ————————————————————▶ lower = middle

False ———————————————————▶ upper = middle
```

Once again we have a test followed by some actions. This time, however, the test doesn't decide whether or not to run a block of code again, but rather which of two blocks of code to run once.

Our test — a Python expression that evaluates to a boolean — is simply:

```
middle**2 < 2.0
```

and if this evaluates to `True` then we change the lower bound:

```
lower = middle
```

and if it evaluates to `False` then we change the upper bound:

```
upper = middle
```

Either way, once one or the other has run we move on and do not return to the test.

Simple example

```
text = input('Number? ')
number = int(text)

if number % 2 == 0:
    print('Even number')
else:
    print('Odd number')

print('That was fun!')
```

ifthenelse1.py

```
$ python3 ifthenelse1.py

Number? 8
Even number
That was fun


$ python3 ifthenelse1.py

Number? 7
Odd number
That was fun
```

Again, we will look at an example that demonstrates just the structure.

There is a script in your home directories called ifthenelse1.py which illustrates the structure on its own.

Mathematical note:

The script tests for a number being even by using the "remainder" operator "%" to calculate the remainder if we divide by 2 and testing for that remainder being 0.

if…then… else… — 1


```
if number % 2 == 0 :
    print('Even number')
else :
    upper = middle
print('That was fun!')
```

The first line of the test looks very similar to the `while` syntax we have already seen. In this case, however, it uses a new keyword: "`if`".

The `if` keyword is followed by the test: a Python expression that evaluates to a boolean.

The line ends with a colon.

if…then… else… — 2

```
if number % 2 == 0 :

    print('Even number')        ⟵ Run if test is **True**
                                    Indentation
else :

    upper = middle

print('That was fun!')
```

The test line is immediately followed by the block of code that is run if the test evaluates as `True`.

Because it is a block of code it is indented by four spaces to mark it as a block. This example has a single line, but the block can be as long as you want.

This block is sometimes called the "then-block" because "if the test is `True` *then* run this block".

if…then… else… — 3

```
if number % 2 == 0 :

␣␣␣␣print('Even number')

else :

␣␣␣␣upper = middle

print('That was fun!')
```

else: keyword

Run if test is **False**

Indentation

After the then-block comes another new keyword, "`else:`". This is not indented and is level with the "`if`" to indicate that it is not part of the then-block.

It is then followed by a second block of code, know as the "else-block". This is the code that is run if the test evaluates as `False`.

Again, because it is a block of code it is indented.

The `else` keyword and its corresponding block are optional. You can do nothing if the test returns `False`. The then-block is compulsory.

if...then... else... — 4

```
if number % 2 == 0 :

␣␣␣␣print('Even number')

else :

␣␣␣␣ upper = middle

print('That was fun!')
```

Run afterwards
regardless of test

After the else-block the script continues. The `print` line is unindented so is not part of the else-block. This line is run regardless of the result of the test.

Our square root example

```
middle = (lower + upper)/2
```
⟵ Before

```
if middle**2 < 2.0 :

    lower = middle

else :

    upper = middle
```
⟵ if… block

```
print(lower, upper)
```
⟵ After

172

Let's return to our square root example.

Here we have the creation of a mid-point x-value followed by an if-test on it:

```
middle = (lower+upper)/2
if middle**2 < 2.0:
```

This switches between two single-line code blocks. If the test evaluates to `True` then the then-block is run:

```
    lower = middle
```

and if it evaluates to `False` then the else-block is run:

```
else:
    upper = middle
```

After one or other is run the `print` statement is always run:

```
print(lower, upper)
```

All we have to do now is to fit it inside our `while` loop.

Progress

if ... :

else:

choice of two
code blocks

␣␣␣␣indentation

before

if *test* :
␣␣␣␣*action$_1$*
␣␣␣␣*action$_2$*
else:
␣␣␣␣*action$_3$*

afterwards

173

Exercise 8

For each script:

☐☐☐☐ Predict what it will do.

☐☐☐☐ Run the script.

☐☐☐☐ Were you right?

`ifthenelse2.py`

`ifthenelse3.py`

`ifthenelse4.py`

5 minutes

174

Back to our example

```
tolerance = 1.0e-15
lower = 0.0
upper = 2.0
uncertainty = upper - lower

while uncertainty > tolerance :

    middle = (lower + upper)/2

    if middle**2 < 2.0 :

        lower = middle

    else :

        upper = middle

    print(lower, upper)
    uncertainty = upper - lower
```

if **starts** indented

Doubly indented

175

So how do we embed an if-test with its two code blocks inside a while-loop as the loop's body?

The body of the while-loop is indented four spaces. So we start the if-test indented four spaces and make its indented blocks doubly indented.

Levels of indentation

```
tolerance = 1.0e-15
lower = 0.0
upper = 2.0
uncertainty = upper - lower

while uncertainty > tolerance :
    middle = (lower + upper)/2          ⟵ 4 spaces
    if middle**2 < 2.0 :
        lower = middle                  ⟵ 8 spaces
    else :
        upper = middle
    print(lower, upper)
    uncertainty = upper - lower
```

176

So if our standard indentation is four spaces then the doubly indented sections are indented eight spaces.

This is a simple example with only two levels of indentation. Python can 'nest' blocks much further than this.

Trying it out

```
tolerance = 1.0e-15
lower = 0.0
upper = 2.0
uncertainty = upper - lower

while uncertainty > tolerance :
    middle = (lower + upper)/2

    if middle**2 < 2.0:
        lower = middle
    else:
        upper = middle

    print(lower, upper)
    uncertainty = upper - lower
```

sqrt1.py

```
$ python3 sqrt1.py

1.0 2.0
1.0 1.5
1.25 1.5
1.375 1.5
1.375 1.4375
1.40625 1.4375
1.40625 1.421875

...

1.414213... 1.414213...
```

☺

The file sqrt1.py in your home directories contains the code as described in the slide. It produces a very nice approximation to the square root of 2.

Script for the square root of 2.0

```
tolerance = 1.0e-15
lower = 0.0
upper = 2.0                                          √2.0
uncertainty = upper - lower

while uncertainty > tolerance :

     middle = (lower + upper)/2

     if middle**2 < 2.0 :                            √2.0

          lower = middle

     else :

          upper = middle

     print(lower, upper)
     uncertainty = upper - lower
```

178

So now we have the script for the square root of 2. The next thing to do is to generalize it to produce square roots of any number.

Input target

```
text = input('Number? ')
number = float(text)


...


    if middle**2 < number :
```

Obviously we have to input the number whose square root we want. We have already seen how to do this and to convert it from a string into a floating point number:

```
text = input('Number? ')
number = float(text)
```
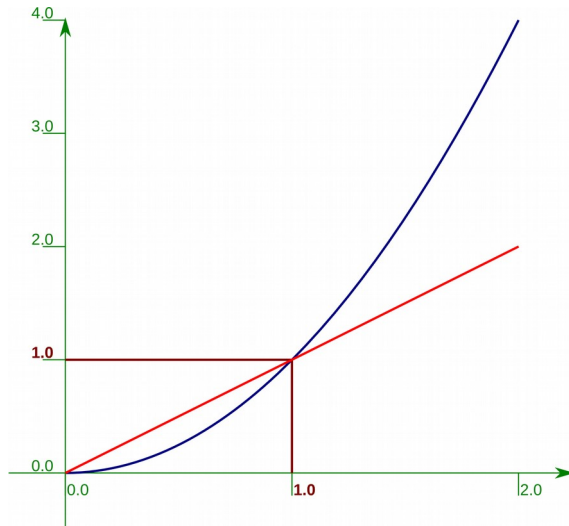
Once we have the number the test

```
middle**2 < 2.0
```

is straightforwardly extended to

```
middle**2 < number
```

Initial bounds?

lower = **?**
upper = **?**

x > 1.0 ⟶ 1.0 < √**x** < **x**

1.0 > **x** ⟶ 1.0 > √**x** > **x**

```
if...then...else...
```

180

We have to set initial bounds for our interval of uncertainty.

This is where it is important that you think about the problem before coding. If the number whose square root is sought is less than 1 then the square root is bigger than the number and less than 1. If it is larger than 1 then its square root is less than the number and bigger than 1.

In Python terms this means we can test for the number being less than 1 and set the bounds accordingly.

Initial bounds

```
if number < 1.0 :
    lower = number
    upper = 1.0
else :
    lower = 1.0
    upper = number
```

It looks like this.

Generic square root script?

```python
text = input('Number? ')
number = float(text)

if number < 1.0:
    lower = number
    upper = 1.0
else:
    lower = 1.0
    upper = number

tolerance = 1.0e-15
uncertainty = upper - lower

while uncertainty > tolerance:
    middle = (lower+upper)/2.0
    if middle**2 < number:
        lower = middle
    else:
        upper = middle

    uncertainty = upper - lower

print(lower, upper)
```

User input

Initialization

Processing

Output

sqrt2.py

This gives us enough of a script to see the overarching structure of a script:

We start with getting the data we need from the outside world. ("input")

Then we set up any initial state we need based on that. ("initialization")

Then we do our processing.

Finally we reveal our results. ("output")

Typically the processing phase takes longest to run, but note that, as here, it is often not the majority of the lines of code.

Negative numbers?

Need to catch negative numbers

```
if number < 0.0:
␣␣␣␣print('Number must be positive!')
␣␣␣␣exit()
```

Quit immediately

"else" is optional

We can improve our code. A step missing from the previous script is "input validation" where we check that the input makes sense. We ought to check that we have not been asked to generate the square root of a negative number.

"Chained" tests

```
text = input('Number? ')
number = float(text)

if number < 0.0:
    print('Number must be positive!')
    exit()

if number < 1.0:
    lower = number
    upper = 1.0
else:
    lower = 1.0
    upper = number

...
```

User input

Input validation

Initialization

184

The input validation phase comes straight after the input itself.

"Chained" tests — syntactic sugar

```
text = input('Number? ')
number = float(text)

if number < 0.0:
    print('Number must be positive!')
    exit()

elif number < 1.0:                    elif: "else if"
    lower = number
    upper = 1.0
else:
    lower = 1.0
    upper = number

...
```

185

`sqrt3.py`

However, it can be integrated with the initialization phase, and often is. After all, if you can't initialize from the input then the input isn't valid.

This leads us to a multi-stage test of the number whose square root we want:

Is it less than 0?

If not, is it less than 1?

If not then…

Python has an extension to the simple `if…else…` test to allow for the "if not then is it…" situation.

"`elif`" introduces a test and a corresponding block of code. The code is called only if the previous `if…` test failed and its own test passes.

Without `elif`...

```
text = input('Number? ')
number = float(text)

if number < 0.0:
    print('Number is negative.')
else:
    if number < 1.0:
        print('Number is between zero and one.')
    else:
        if number < 2.0:
            print('Number is between one and two.')
        else:
            if number < 3.0:
                print('Number is between two and three.')
            else:
                print('Number is three or more.')
```

186

Stacked clauses get unwieldy

To take an extreme example, consider this multi-level test. The continual nesting inside the `else` clauses causes the whole script to drift to the right.

With `elif`...

```
text = input('Number? ')
number = float(text)

if number < 0.0:
    print('Number is negative.')
elif number < 1.0:
    print('Number is between zero and one.')
elif number < 2.0:
    print('Number is between one and two.')
elif number < 3.0:
    print('Number is between two and three.')
else:
    print('Number is three or more.')
```

187

Applying `elif` causes everything to slide back into place.

Progress

Nested structures

Chained tests

Testing inputs to scripts

```
while … :
    if … :


if … :
    …
elif … :
    …
elif … :
    …
else:
    …


exit()
```

188

Exercise 9

`exercise9.py`

1. Edit the square root script to catch negative numbers.

2. Edit the square root script to ask for the tolerance.

3. Edit the square root script to catch negative tolerances.

10 minutes

The minimum tolerance used in this example will work for target values up to a limit. In practice the minimum possible tolerance scales with the target square root, but this is a course on Python, not numerical analysis.

Comments

We have written our first real Python script

What did it do?

Why did it do it?

Need to annotate the script

`sqrt3.py` is a real program.

Now imagine you pass it to someone else or put it away for twelve months and come back to it forgetting how you wrote it in the first place.

Chances are that the reader of your script might like some hints as to what it is doing and why.

"Comments" in computer programs are pieces of text that describe what ios going on without getting in the way of the lines of code that are executed.
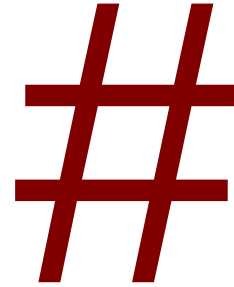
Python comment character

The "hash" character

a.k.a. "pound", "number", "sharp"

Lines starting with "#" are ignored

Partial lines starting "#" are ignored

\#

Used for annotating scripts

191

Python, in common with most other scripting languages, uses the hash character to introduce comments. The hash character and everything beyond it on the line is ignored.

(Strictly speaking the musical sharp character "♯" is not the same as "#" but people get very sloppy with similar characters these days. This isn't at all relevant to Python but the author is an annoying pedant on the correct use of characters. And don't get him started on people who use a hyphen when they should use an en-dash or em-dash.)

Python commenting example

```
# Script to calculate square roots by bisection
# (c) Bob Dowling 2012. Licensed under GPL v3.0
text = input('Number?␣')
number = float(text)  # Need a real number

# Test number for validity,
# set initial bounds if OK.
if number < 0.0:
␣␣␣␣print('Number must be non-negative!')
␣␣␣␣exit()
elif number < 1.0:
␣␣␣␣lower = number
␣␣␣␣upper = 1.0
else:
␣␣␣␣lower = 1.0
␣␣␣␣upper = number
```

192

This is what a commented script looks like.

On a *real* Unix system…

```
#!/usr/bin/python3

# Script to calculate square roots by bisection
# (c) Bob Dowling 2012. Licensed under GPL v3.0
text = input('Number?␣')
number = float(text)  # Need a real number
```

Magic line for executable files

```
$ fubar.py
```
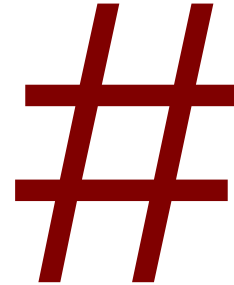instead of
```
$ python3 fubar.py
```

You may encounter a "hash pling" first line in many imported Python scripts. This is part of some "Unix magic" that lets us simplify our command lines. We can't demonstrate it here because the MCS file server doesn't support Unix semantics.

Progress

Comments

"#" character

#

194

Exercise 10

Comment your square root script from exercise 9.

2 minutes

Recap: Python types so far

Whole numbers	-127
Floating point numbers	3.141592653589793
Complex numbers	(1.0 + 2.0j)
Text	`'The cat sat on the mat.'`
Booleans	True False

196

We are about to introduce a new Python type, so we will take a moment to remind ourseoves of the various Python types we have met already.

Lists

```
[ 'hydrogen', 'helium', 'lithium', 'beryllium',
'boron', …, 'thorium', 'protactinium', 'uranium' ]


[ -3.141592653589793, -1.5707963267948966,
0.0, 1.5707963267948966, 3.141592653589793 ]


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

The new Python type we are going to meet is called a "list".

What is a list?

hydrogen, helium, lithium, beryllium, …, protactinium, uranium

A *sequence* of values   The names of the elements

Values stored in order   Atomic number order

Individual value identified   "helium" is the name of the
by position in the sequence   second element

198

So what is a list?

A list is simply a sequence of values stored in a specific order with each value identified by its position in that order.

So for an example consider the list of names of the elements up to uranium.

What is a list?

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59

A *sequence* of values

The prime numbers
less than sixty

Values stored in order

Numerical order

Individual value identified
by position in the sequence

7 is the fourth prime

Or the list of primes up to 60.
Note that a list must be finite.

Creating a list in Python

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

A literal list

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

The whole list

```
>>> type(primes)
<class 'list'>
```
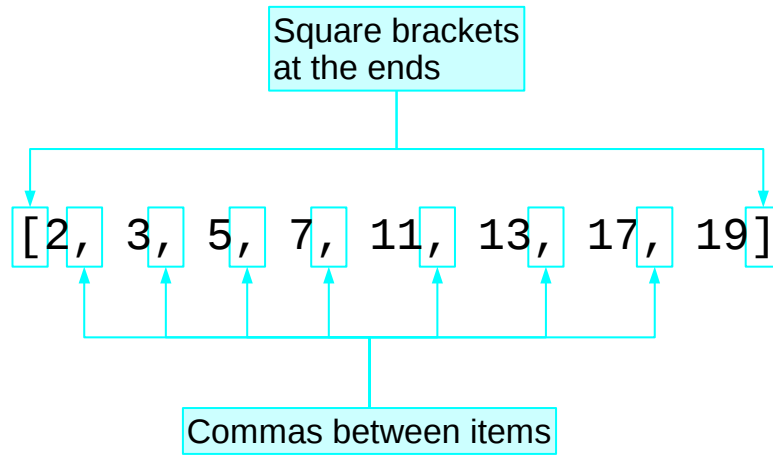
A Python type

200

So how might we do this in Python?

We will create a list in Python of the primes less than 20. We can do this in a single line as shown.

A list in Python is a single Python object, albeit with multiple contents, and has its own type, unsurprisingly called "`list`".

How Python presents lists

Square brackets
at the ends

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

Commas between items

201

Python presents (and accepts) lists as a series of values separated by commas, surrounded by square brackets.

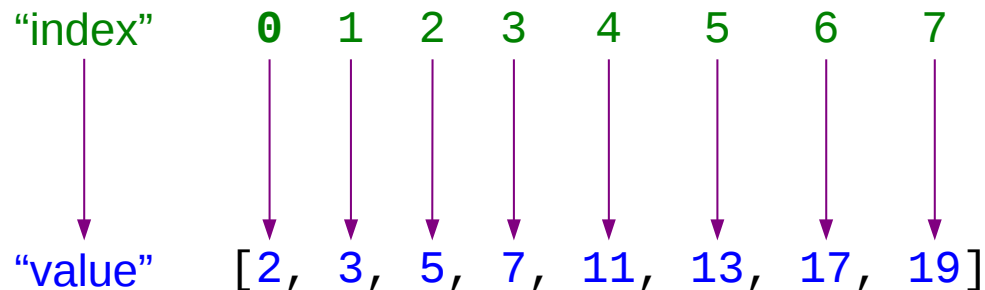# Square brackets

`primes = [2, 3, 5, 7, 11]`   Literal list

202

We are going to meet square brackets used fr a lot of things so I will build up a summary slide of their various uses. Here is use 1.

Python counts from **zero**

"index"   **0**   1   2   3   4   5   6   7

"value"   [2, 3, 5, 7, 11, 13, 17, 19]

We still need to get at individual items in the list. Each is identified by its position in the list.

Python, in common with many programming languages (but not all) starts its count from *zero*. The leading element in the list is "item number zero". The one that follows it is "item number one" and so on. This number, the position in the list counting from zero, is called the "*index*" into the list. The plural of "index" is "indices".

To keep yourself sane, we strongly recommend the language "item number 3" instead of "the fourth item".

Looking things up in a list

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

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

index

```
>>> primes[0]
2
```

square brackets

```
>>> primes[6]
17
```

So, how do we get "item number 5" from a list?

We can follow the list (or, more usually, a name attached to the list) with the index in square brackets.

Square brackets

```
primes = [2, 3, 5, 7, 11]
```
Literal list

```
primes[3]
```
Index into list

205

And this is the second use of square brackets.

Counting from the end

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

```
          0   1   2   3   4    5    6    7
          ↓   ↓   ↓   ↓   ↓    ↓    ↓    ↓
         [2,  3,  5,  7,  11,  13,  17,  19]
          ↑   ↑   ↑   ↑   ↑    ↑    ↑    ↑
         -8  -7  -6  -5  -4   -3   -2   -1
```
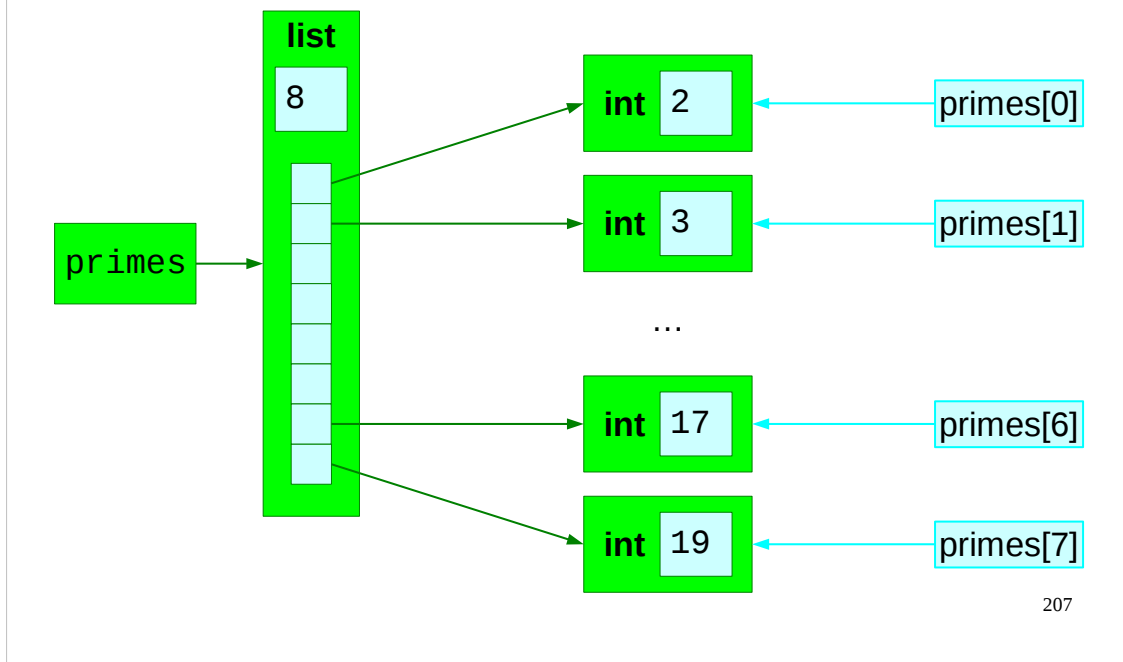
getting at the last item

```
>>> primes[-1]
19
```

Python has a trick for getting at the last element of the list. Negative indices are also valid and count backwards from the end of the list. Typically the only case of this that is used in paractice is that the index -1 gets the last element of the list.

Inside view of a list

list		
8		**int** 2 ← primes[0]
	→ **int** 2	
	→ **int** 3	primes[1]
primes →	...	
	→ **int** 17	primes[6]
	→ **int** 19	primes[7]

207

We've seen these box diagrams for simple Python types already. The structures for lists are a little more complicated but only a little. The list type records how long it is and an ordered set of references to the actual objects it contains.

Length of a list

list

8 → 8

0

primes

7

>>> **len(primes)**

len() function:
length of list

Maximum
index is 7

208
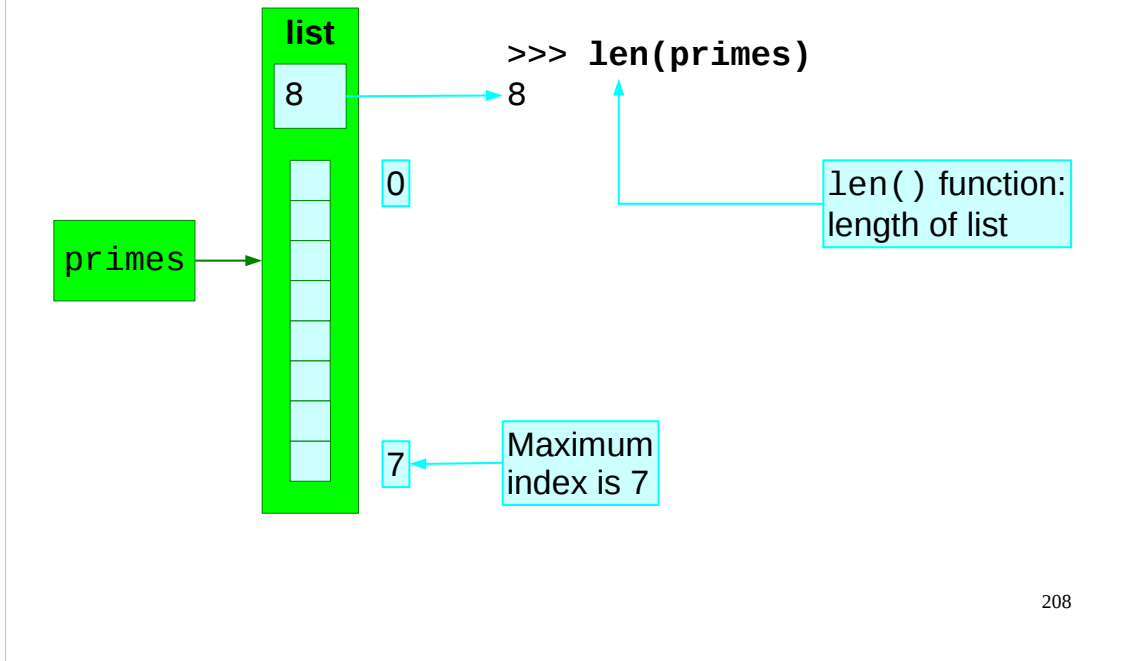
Note that the length of a list is the number of items it contains. The largest legitimate index is one less than that because indices count from zero.

Changing a value in a list

```
>>> data = ['alpha', 'beta', 'gamma']    The list

>>> data[2]                              Initial value
'gamma'

>>> data[2] = 'G'                        Change value

>>> data[2]                              Check change
'G'

>>> data                                 Changed list
['alpha', 'beta', 'G']
```
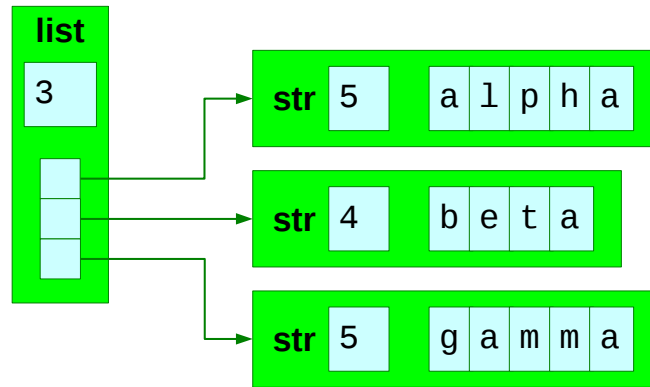
209

So far we have created lists all in one go by quoting a literal list. We can use the indexing notation (square brackets) to *change* items in a list too.

Changing a value in a list — 1

Right to left

```
>>> data = ['alpha', 'beta', 'gamma']
```

list

3

str 5   a l p h a

str 4   b e t a
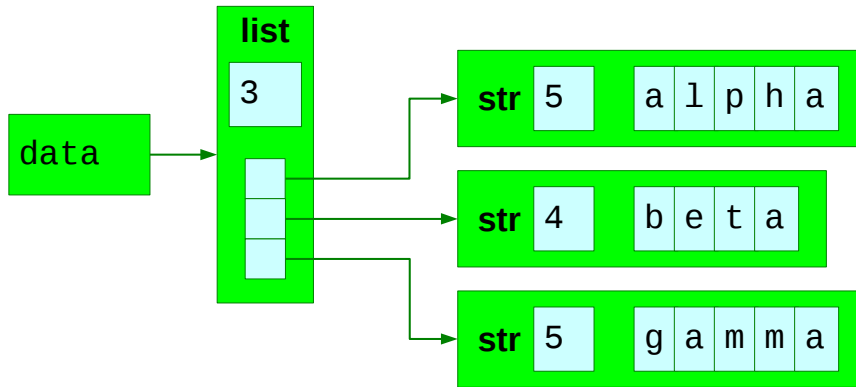
str 5   g a m m a

210

We can track what happens in that example in some detail. We will start with the first line, defining the initial list.

As ever, Python assignment is done right to left. The right hand side is evaluated as a list of three items, all of themstrings.

Changing a value in a list — 2
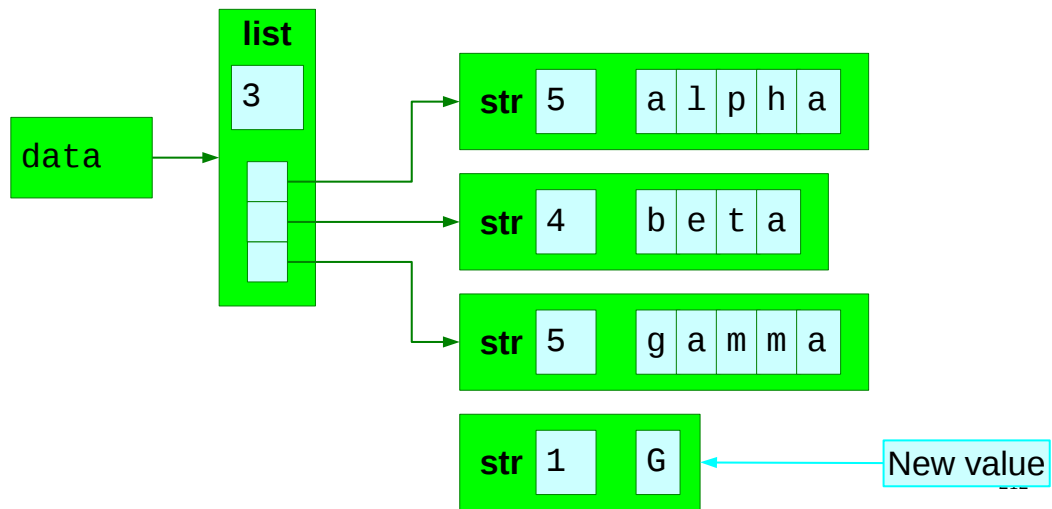
```
>>> data = ['alpha', 'beta', 'gamma']
```


211

This then has the name "`data`" attached to it.

Changing a value in a list — 3

```
>>> data[2] = 'G'
```

list

3

data

str 5   a l p h a

str 4   b e t a
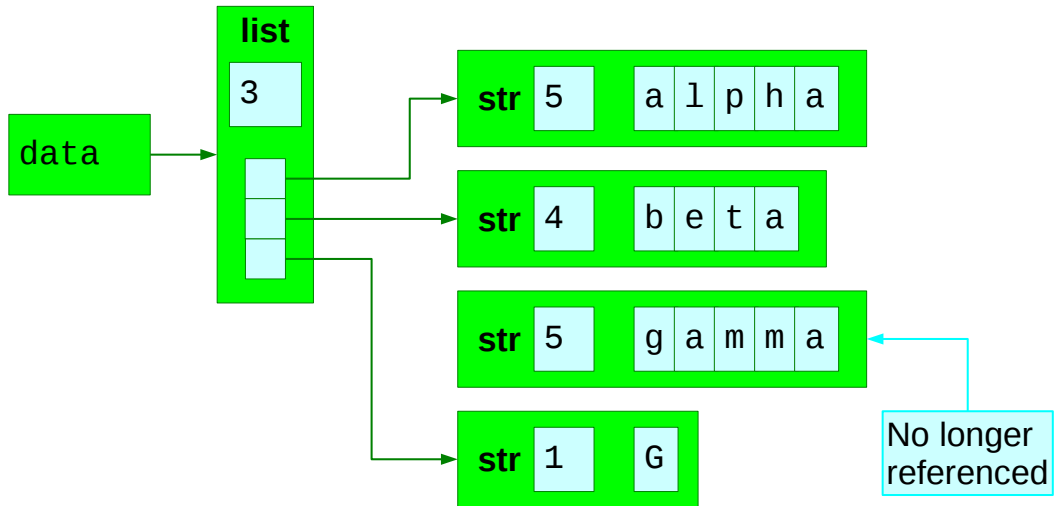
str 5   g a m m a

str 1   G   New value

Now we come to the second line which changes one of these list items.

The right hand side is evaluated as a string containing a single characters. That object gets created.

Changing a value in a list — 4

```
>>> data[2] = 'G'
```

list

3

data

str 5   a l p h a

str 4   b e t a

str 5   g a m m a

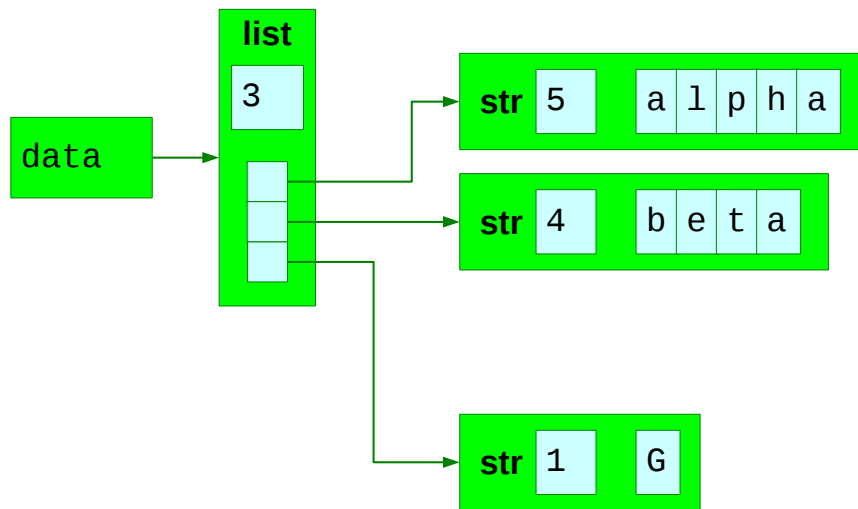str 1   G

No longer referenced

The assignment causes the reference within the string to be changed to refer to the new string and to stop referring to the previous one, "gamma".

In this case there are now no references to the "gamma" string.

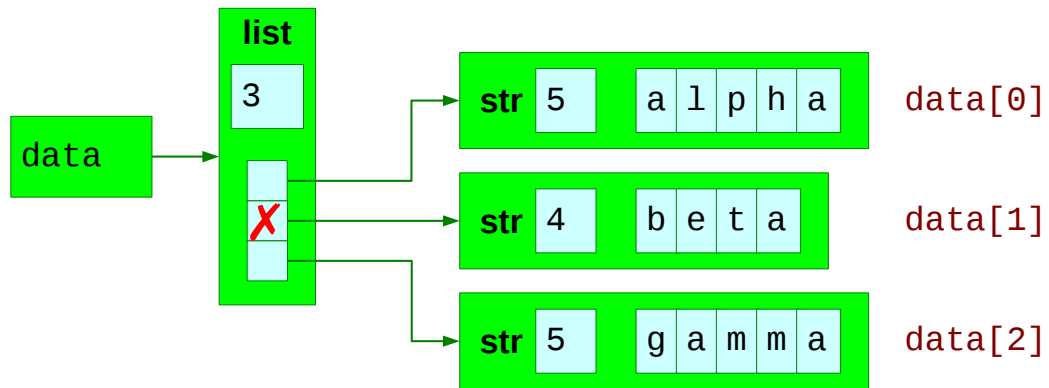# Changing a value in a list — 5

```
>>> data[2] = 'G'
```


214

Python then clears out the memory used for that old string so that it can reuse it for something else. This process is called "garbage collection" in computing.

Removing an entry from a list — 1
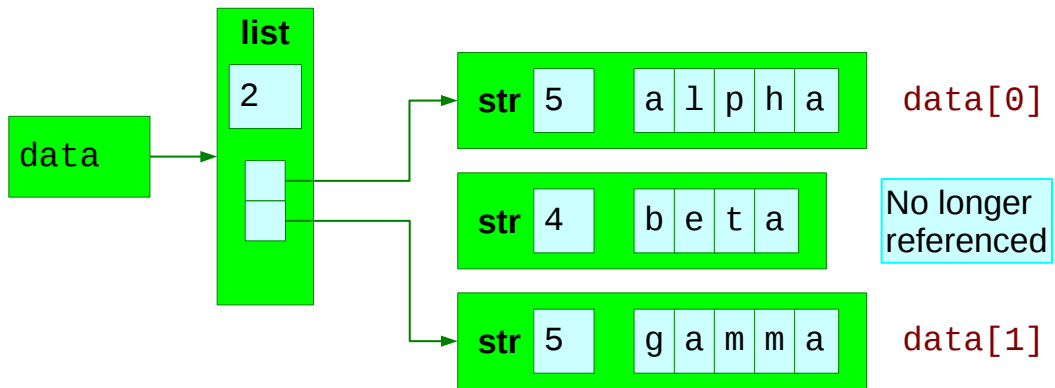
```
>>> del data[1]
```


215

We can remove entries from the list too with the "del" keyword, just as we removed names. The del keyword removes the reference from the list.

Removing an entry from a list — 2

```
>>> del data[1]
```

list

2

data

str 5 | a | l | p | h | a   `data[0]`

str 4 | b | e | t | a   No longer referenced

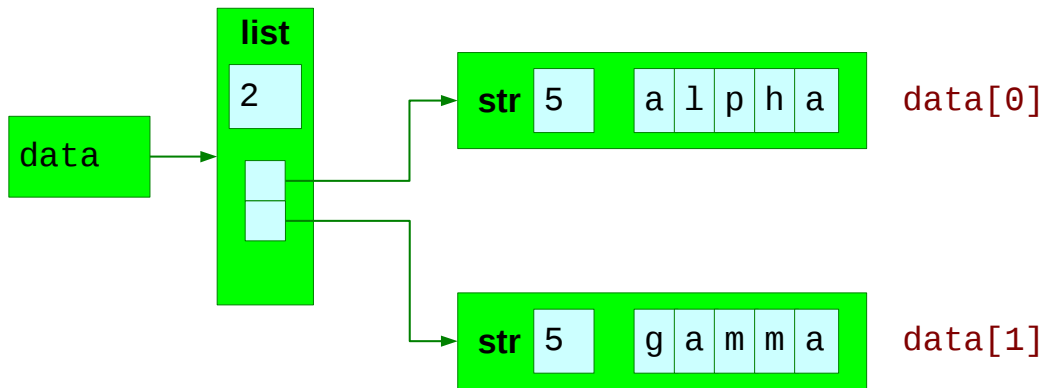**str** 5 | g | a | m | m | a   `data[1]`

216

This leaves the string "`beta`" no longer referenced by anything.

Removing an entry from a list — 3

```
>>> del data[1]
```


And garbage collection kicks in again.

Running off the end

list

8

primes

int 2

int 3

...

int 17

int 19
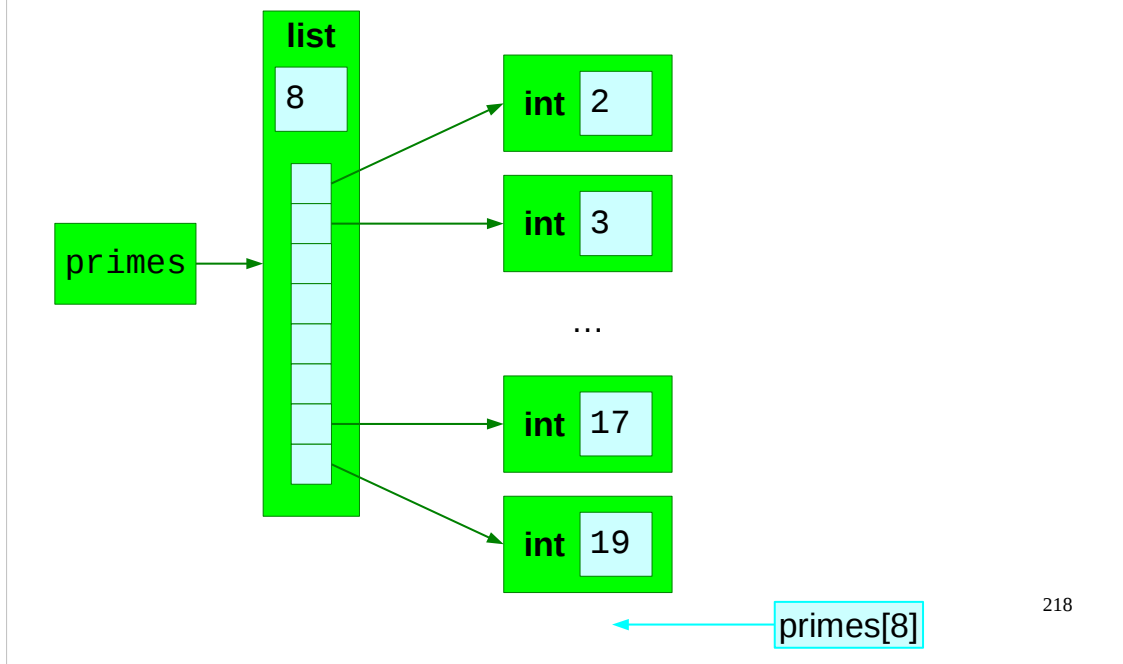
primes[8]

We have remarked on a couple of occasions that the largest valid index is a number one less than the length of the list.

So what happens if you ask for an index greater than the largest legal value?

Running off the end

```
>>> len(primes)
8

>>> primes[7]
19

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

Type of error   Description of error

219

You get an error unsurprisingly.

The type of the error is an "IndexError" — something went wrong with an index.

The error message specifies that the index asked for was outside the valid range.

Running off the end

```
>>> primes[8] = 23

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Same type
of error

Similar description of error
but with "assignment"

Note that we can't use indicies beyond the limit to extend a list either.

Progress

Lists	`[2, 3, 5, 7, 11, 13, 17, 19]`
index	`primes[4]`
Count from zero	`primes[0]`
Deletion	`del primes[6]`
Length	`len(primes)`
Over-running	`primes[8]`

221

Exercise 11

Track what the value of `numbers` is at each stage of this sequence of instructions.

1   `>>> numbers = [5, 7, 11, 13, 17, 19, 29, 31]`

2   `>>> numbers[1] = 3`

3   `>>> del numbers[3]`

4   `>>> numbers[3] = 37`

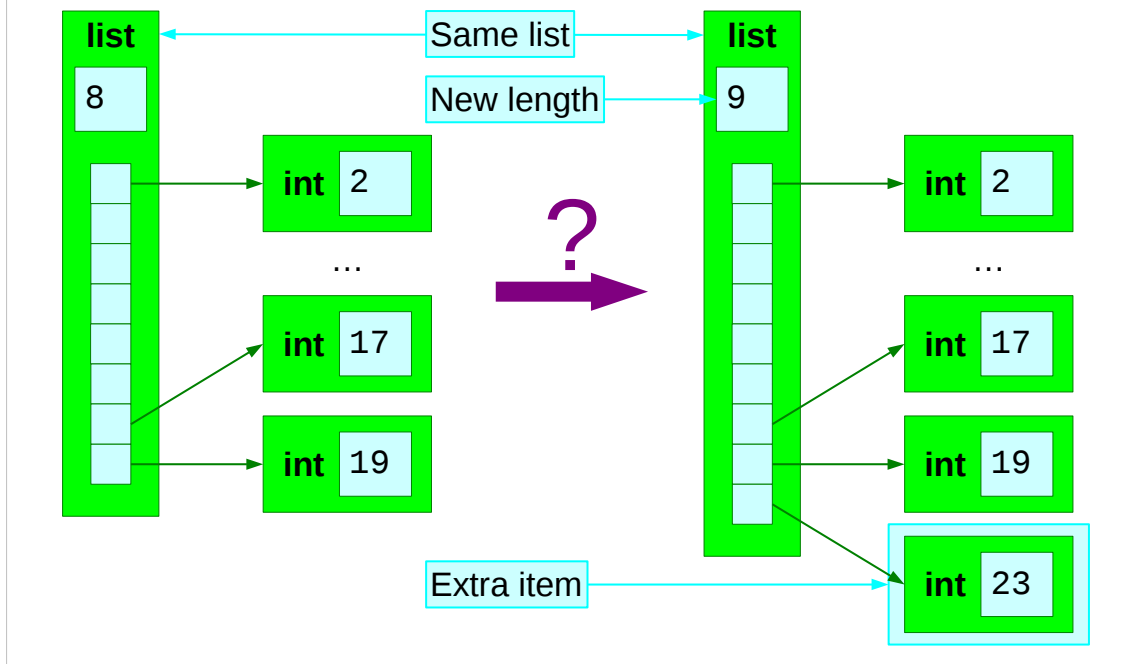5   `>>> numbers[4] = numbers[5]`

5 minutes

222

Do this by hand. After each line, work out what you think numbers will be and then print it to see if you were right.

To find out, launch an interactive `python3` and run each line followed by the line `print(numbers)`.

Alternatively, the script `exercise11.py` does it in a script.

How can we add to a list?

So, how can we extend a list?

Appending to a list

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


>>> primes.append(23)
```

A function built into a list
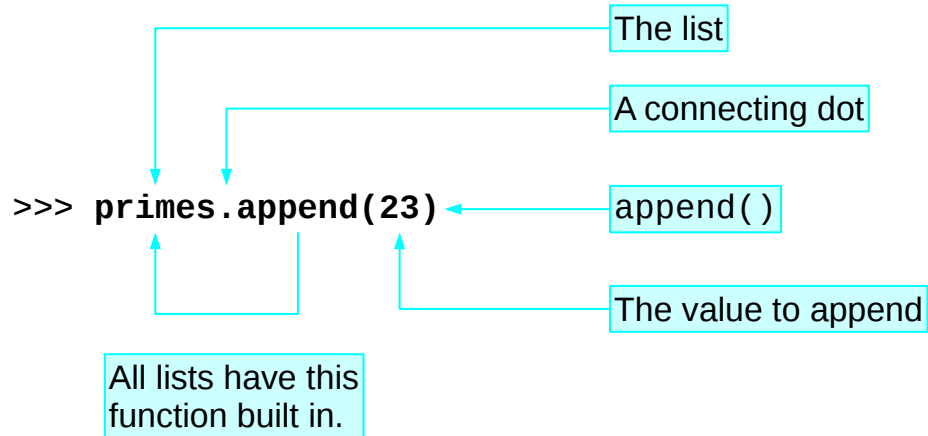
```
>>> primes
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

The list is now updated

This is the Python syntax for appending an item to the end of a list. You won't recognise the syntax; it is something new.
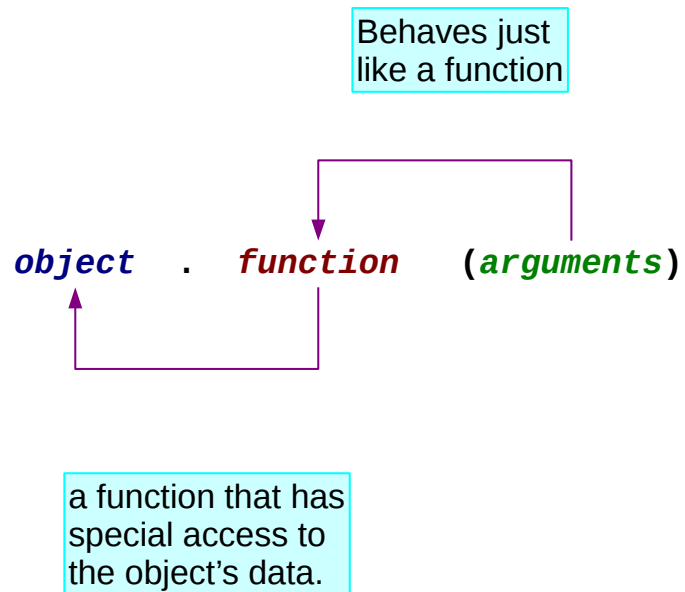
primes.append() ?

```
                                                    ┌──────────┐
                                          ┌─────────│ The list │
                                          │         └──────────┘
                                          │         ┌────────────────┐
                                  ┌───────│─────────│ A connecting dot│
                                  │       │         └────────────────┘
                                  ▼       ▼         ┌──────────┐
>>> primes.append(23) ◄───────────────────────────│ append() │
          ▲           ▲                            └──────────┘
          │           │                            ┌────────────────────┐
          └───────────┘          ┌─────────────────│ The value to append│
                                 │                 └────────────────────┘
          ┌────────────────┐
          │ All lists have this│
          │ function built in. │
          └────────────────┘
```

So we need to look at this new construction.

We have the list, "`primes`", followed by a dot which acts as a connector. This is followed by the name of a function, "`append`". This function is not a standard Python function like `print()` or `len()`. Instead it is a function that is "built in" to the list itself. The list has its own function which appends to that list.

Alternatively, think of "`primes.append()`" as a function that appends to `primes`.

"Methods"

Behaves just
like a function

***object* . *function* (*arguments*)**

a function that has
special access to
the object's data.

These built in functions are called "methods" or, more precisely, "methods of the object" are used all over Python and are a general concept across an entire type of programming called "object oriented programming".

Using the append() method

```
>>> print(primes)
[2, 3, 5, 7, 11, 13, 17, 19]

>>> primes.append(23)

>>> primes.append(29)

>>> primes.append(31)

>>> primes.append(37)

>>> print(primes)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

The function doesn't return any value.

It modifies the list itself.

227

We can use the append() method repeatedly to extend the list as far as we want.

Other methods on lists: `reverse()`

```
>>> numbers = [4, 7, 5, 1]

>>> numbers.reverse()
```
The function doesn't return any value.

```
>>> print(numbers)

[1, 5, 7, 4]
```
It modifies the list itself.

append() is not the only method built into lists.

For example `reverse()` takes the list and reverses its contents.

Note that it doesn't return a reversed list as a value; t doesn't return anything at all.

It silently reverses the content of the list itself.

Also note that it takes no argument; the brackets on the end of the function are empty.

Other methods on lists: `sort()`

```
>>> numbers = [4, 7, 5, 1]

>>> numbers.sort()

>>> print(numbers)

[1, 4, 5, 7]
```

The function does not return the sorted list.

It sorts the list itself.

Numerical order.

Similarly, the `sort()` method doesn't return a sorted list but silently sorts the list internally.

Other methods on lists: `sort()`

```
>>> greek = ['alpha', 'beta', 'gamma', 'delta']


>>> greek.sort()


>>> print(greek)

['alpha', 'beta', 'delta', 'gamma']
```
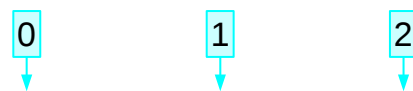
Alphabetical order
of the *words*.

230

More or less any type can be sorted. Text sorting carries all the cautions about the complexities of collation that we covered under comparisons.
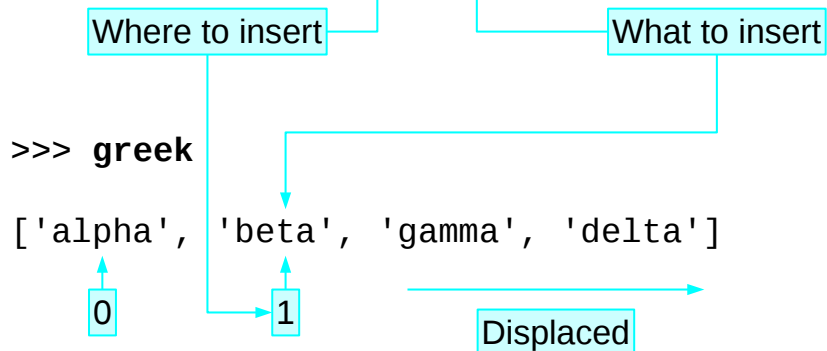
Other methods on lists: `insert()`

```
            0          1          2
>>> greek = ['alpha', 'gamma', 'delta']

>>> greek.insert(1, 'beta')
```

Where to insert ← → What to insert

```
>>> greek

['alpha', 'beta', 'gamma', 'delta']
        0        1        Displaced
```

The `append()` method sticks an item on the end of a list. If you want to insert an item elsewhere in the list we have the `insert()` method.

The `insert()` method takes two arguments:

The first is the item to be inserted.

The second is in index where it should go. This does not replace the original item but rather "shuffles up" all the items beyond it by one place to make room.

Other methods on lists: `remove()`

```
>>> numbers = [7, 4, 8, 7, 2, 5, 4]

>>> numbers.remove(8)          Value to remove

>>> print(numbers)
[7, 4, 7, 2, 5, 4]


c.f.  del numbers[2]          Index to remove
```

232

There is a `remove()` method.

This is passed a value to remove from the list. It then removes that value from the list, wherever it is in the list.

Contrast with with `del` where you had to know the index to remove.

Other methods on lists: `remove()`

```
>>> print(numbers)

[7, 4, 7, 2, 5, 4]
```

There are two instances of 4.

```
>>> numbers.remove(4)


>>> print(numbers)

[7, 7, 2, 5, 4]
```

Only the first instance is removed

233

If the value appears more than once in a list then only the *first* instance is removed.

Trying to remove something that isn't there will lead to an error.

What methods are there?

```
>>> help(numbers)

Help on list object:

class list(object)
...
 |  append(...)
 |      L.append(object) -- append object to end
...
```

Pagination:		
	␣	next page
	B	back one page
	Q	quit

That's a lot of methods, and it's only some of them. How can we know all of them?

You can always ask for help on any Python object and you will be told all about the methods it possesses. It is a very formal documentation but the information is there.

Incidentally, Python uses a program to paginate its help output. press the space bar to move on one page, "B" to move **b**ack a page and "Q" to **q**uit.

Help on a single method

```
>>> help(numbers.append)

Help on built-in function append:

append(...)
    L.append(object) -- append object to end
```

You can also get help on a single method which is often simpler to deal with.

Sorting a list *redux*

```
>>> greek = ['alpha', 'beta', 'gamma', 'delta']

>>> greek.sort()
```
Recall: `greek.sort()` sorts the list "in place".
```
>>> print(greek)

['alpha', 'beta', 'delta', 'gamma']
```

236

We noted that the sort() method sorts the list itself. Experience shows that sorting is one of those operations where people want a sorted copy of the list quite often.

Sorting a list *redux*: "sorted()"

```
>>> greek = ['alpha', 'beta', 'gamma', 'delta']

>>> print(sorted(greek))
['alpha', 'beta', 'delta', 'gamma']
```

`sorted()` function returns a sorted list...

```
>>> print(greek)
['alpha', 'beta', 'gamma', 'delta']
```

...and leaves the list alone

237

To assist with this, Python 3 offers a standalone function called `sorted()` which makes a copy of the list and sorts that copy, leaving the original unchanged.

Adding to a list *redux*: "+"

```
>>> primes

[2, 3, 5, 7, 11, 13, 17, 19]
```
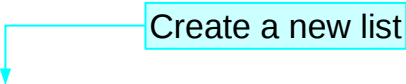
Concatenation operator   List to add

```
>>> primes + [23, 29, 31]
```
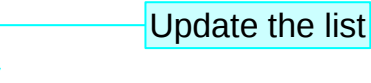
`[2, 3, 5, 7, 11, 13, 17, 19,` `23, 29, 31]`

238

The list method we saw first appended a single item to the end of a list. What happens if we want to add a whole list of items at the end?

In this regard, lists are like strings. The "+" operator performs concatenation and creates a new list which is one concatenated after the other.
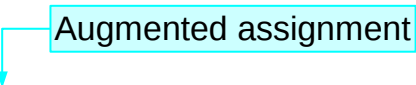
Concatenation

Create a new list

```
>>> newlist = primes + [23, 29, 31]
```

Update the list

```
>>> primes  = primes + [23, 29, 31]
```

Augmented assignment

```
>>> primes += [23, 29, 31]
```
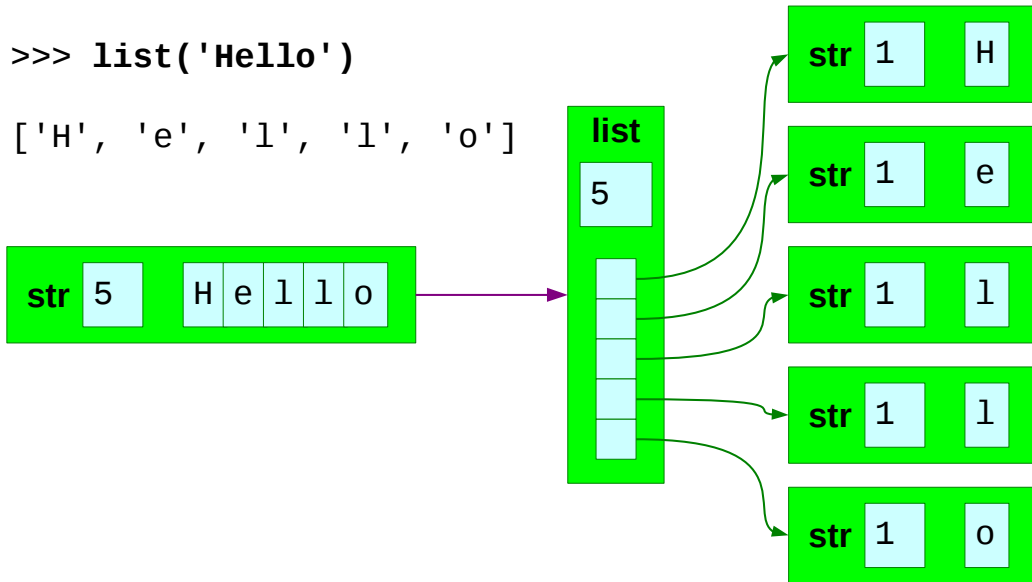
239

We can use this to update a list in place. Note that the augmented assignment operator "+=" also works and is more than syntactic sugar this time. It is actually more efficient than the long hand version because it updates the list in place rather than creating a new one.

Creating lists from text — 1

```
>>> list('Hello')

['H', 'e', 'l', 'l', 'o']
```

We ought to look at a couple of ways to create lists from text.

The first is to simply convert a string into a list with the `list()` function.

(As with all Python types, there is a function of the same name as the type that converts into the type.)
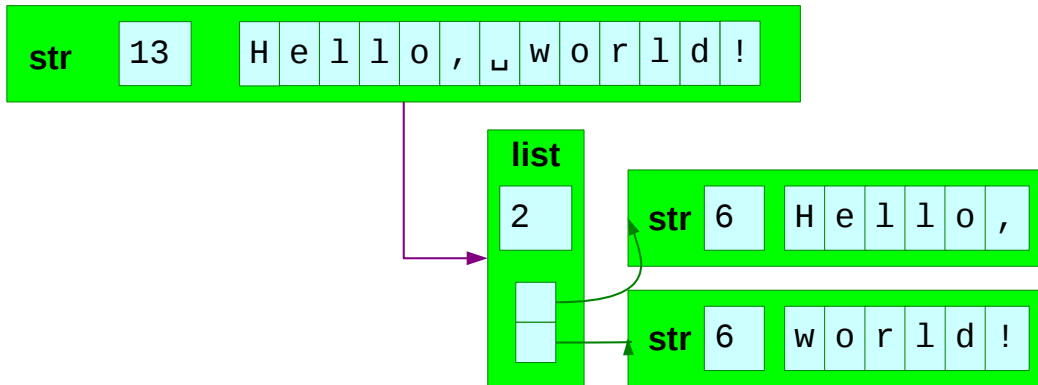
Applying `list()` to a string gives a list of the characters in the string.

Creating lists from text — 2

Built in method

```
>>> 'Hello, world!'.split()

['Hello,', 'world!']
```

Splits on spaces

| str | 13 | | H | e | l | l | o | , | ␣ | w | o | r | l | d | ! |

list

2

| str | 6 | | H | e | l | l | o | , |

| str | 6 | | w | o | r | l | d | ! |

The string type has methods of its own. One of these is split() which returns a list of the components of the string as separated by white space.

The split() method can take an argument identifying other characters to split on. If you want to get into more complex splitting of text we recommend you investigate regular expressions or format-specific techniques (e.g. for comma-separated values.

Progress

"Methods"   `object`.`method`(`arguments`)

  `append(item)`
  `reverse()`
  `sort()`
  `insert(index,item)`
  `remove(item)`

Help   `help(`*object*`)`
                             `help(`*object*.*method*`)`

Sorting   *list*`.sort()`
                             `sorted(`*list*`)`

Concatenation

  `+`   `[1,2,3] + [4,5,6]`
  `+=`   `primes += [29, 31]`

Exercise 12

5 minutes

243

Is an item in a list? — 1

```
>>> odds = [3, 5, 7, 9]
```
Does not include 2

```
>>> odds.remove(2)
```
Try to remove 2

```
Traceback (most recent call last):
```
Hard error
```
    File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

✗

x must be in the list before it can be removed

Recall that a list's remove() method will give an error if the value to be removed is not in the list to start with.

We need to be able to test for whether an item is in a list or not.

Is an item in a list? — 2

```
>>> odds = [3, 5, 7, 9]

>>> 2 in odds

False

>>> 3 in odds

True

>>> 2 not in odds

True
```

Python uses the keyword "in" for this purpose. It can be used on its own or as part of "not in". "*value* in *list*" evaluates to a boolean: True or False.

Precedence

First

x**y   -x   +x   x%y   x/y   x*y   x-y   x+y

x==y   x!=y   x>=y   x>y   x<=y   x<y

x not in y   x in y

not x   x and y   x or y   The list now contains
                          every operator we
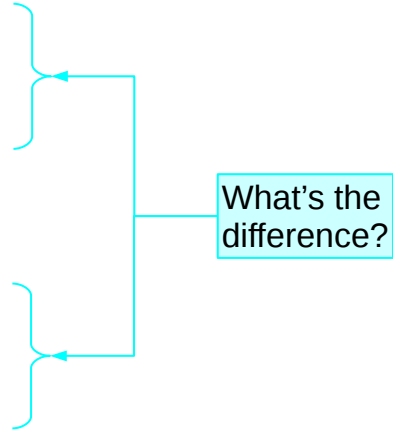   Last   meet in this course.

246

These operators fit naturally into the order of precedence. While Python does contain other operators that belong in this list, we will not be meeting them in this introductory course.

Safe removal

```
if number in numbers :

    numbers.remove(number)



while number in numbers :

    numbers.remove(number)
```

What's the difference?

We now have a safe way to remove values from lists, testing before we remove them.

Quick question: What's the difference between the two code snippets in the slide?

Working through a list — 1

e.g. Printing each element on a line

```
['The', 'cat', 'sat', 'on', 'the', 'mat.']

The
cat
sat
on
the
mat.
```

There is an obvious thing to want to do with a list, and that is to work through each item in a list, in order, and perform some operation or set of operations on each item.

In the most trivial case, we might want to print each item. The slide shows a list of strings, probably from the `split()` of a string. How do we print each item one after the other?

Working through a list — 2

e.g. Adding the elements of a list

```
[45, 76, -23, 90, 15]
```

203

What is the sum of an empty list?   [ ] ⟶ ?

Alternatively, we might want to accumulate the items in a list in some way. For example, we might want to sum the numbers in a list. This is another example of applying an operation to each item in a list. This time the operation is folding the list items' values into some final result.

In this case we would probably need an initial value of the result that it takes before any items get folded into it. What is the sum of an empty list? Zero? Is that an integer zero, a floating point zero, or a complex zero?

Working through a list — 3

e.g. Squaring every number in a list

```
[4, 7, -2, 9, 1]
```



```
[16, 49, 4, 81, 1]
```

Finally, we might want to convert one list into another where each item in the output list is the result of some operation on the corresponding item in the input list.

The "for loop" — 1

name of list

list

```
words = ['The', 'cat', 'sat', 'on', 'the', 'mat.']



for word in words :
```

A new Python
looping construct

```
    print(word)
```

`print`: What we want
to do with the list items.

Python has a construct precisely for stepping through the elements of a list. This is the third and final construct we will meet in this course. We have already seen `if`… and `while`… in this course. Now we meet `for`….

This a looping construct, but rather than repeat while a test evaluates to `True`, it loops once for each item in a list. Furthermore it defines a name which it attaches to one item after another in the list as it repeats the loop.

The "for loop" — 2

```
words = ['The', 'cat', 'sat', 'on', 'the', 'mat.']
```

keywords

```
for word in words :
    print(word)
```

colon followed by an indented block

We'll look at the expression one step at a time.

The expression is introduced by the "`for`" keyword.

This is followed by the name of a variable. We will return to this in the next slide.

After the name comes the keyword "`in`". We have met this word in the context of lists before when we tested for an item's presence in a list. This is a different use. We aren't asking if a specific value is in a list but rather we are asserting that we are going to be processing those values that *are* in it.

After this comes the list itself, or more often the name of a list. All it has to be is an expression that evaluates to a list.

The line ends with a colon.

The lines following the colon are indented marking the block of code that is to be run once for each item in the list.

The "for loop" — 3

```
words = ['The', 'cat', 'sat', 'on', 'the', 'mat.']
```

Defining the **loop variable**

```
for word in words :

     print(word)
```
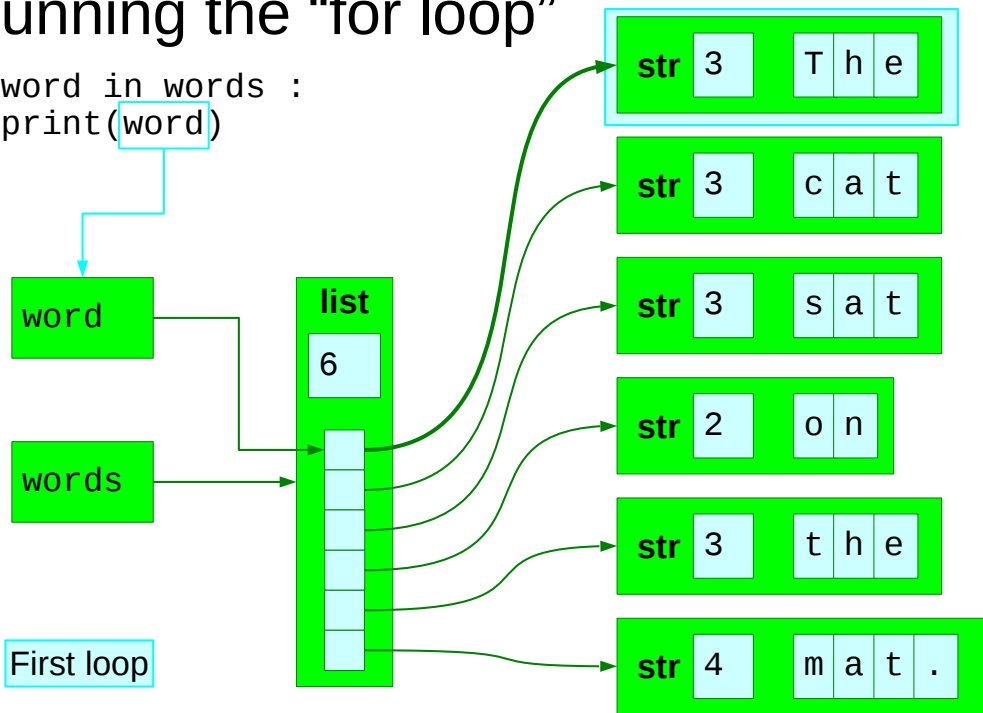
Using the loop variable

Now let's return to the name between "`for`" and "`in`".

This is called the "loop variable". Each time the loop block is run this name is attached to on item of the list. Each tim the loop is run the name is attached to the next item in the list. The looping stops after the name has been attached to the last item in the list.

Running the "for loop"

```
for word in words :
    print(word)
```

list

word

words

First loop

str 3   T h e

str 3   c a t

str 3   s a t

str 2   o n

str 3   t h e

str 4   m a t .

6

The first time the loop is run the "loop variable" name is attached to the first element in the list.

Running the "for loop"

```
for word in words :
    print(word)
```


word

words

list

6

Second loop

str 3   T h e

str 3   c a t

str 3   s a t

str 2   o n

str 3   t h e

str 4   m a t .
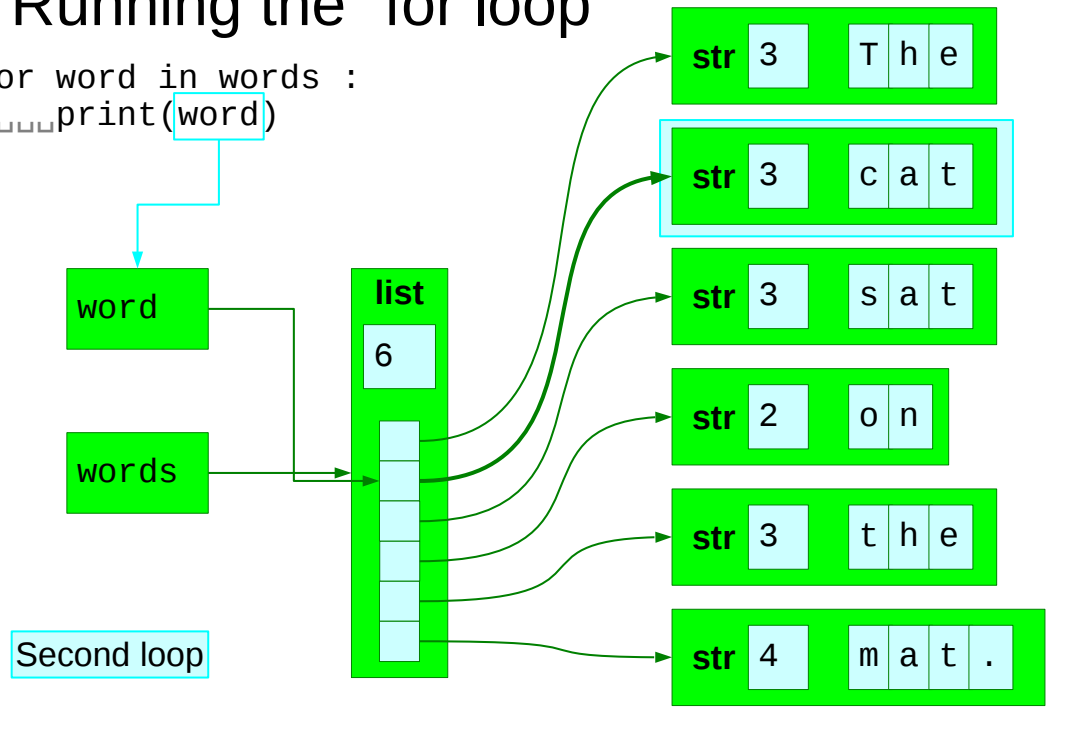
The loop gets run again, but this time the loop variable name is attached to the next element of the list.

Running the "for loop"

```
for word in words :
    print(word)
```
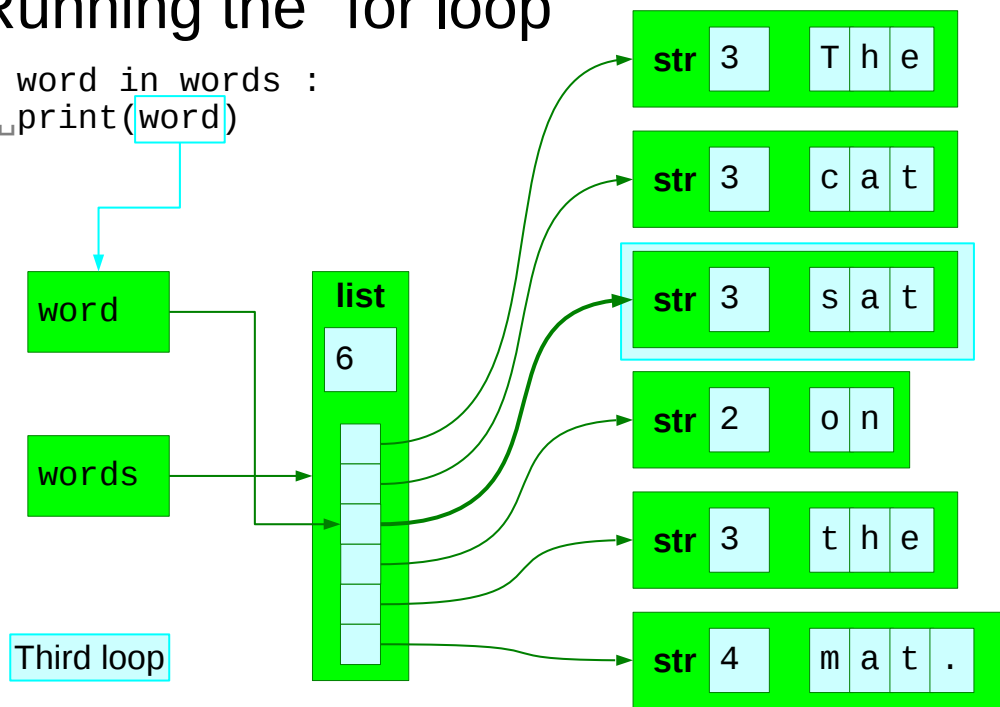
word

words

Third loop

list

6

str 3   T h e

str 3   c a t

str 3   s a t

str 2   o n

str 3   t h e

str 4   m a t .

And the next…

Running the "for loop"

```
for word in words :
    print(word)
```

word	

words	

list

6

str 3   T h e

str 3   c a t

str 3   s a t

str 2   o n

str 3   t h e

str 4   m a t .

Fourth loop

…and the next…

Running the "for loop"

```
for word in words :
    print(word)
```

word

words

Fifth loop

list

6

str	3		T	h	e

str	3		c	a	t

str	3		s	a	t

str	2		o	n

str	3		t	h	e

str	4		m	a	t	.

…and the next…

Running the "for loop"

```
for word in words :
    print(word)
```

list

6

| str | 3 | | T | h | e |

| str | 3 | | c | a | t |

| str | 3 | | s | a | t |

| str | 2 | | o | n |

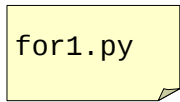| str | 3 | | t | h | e |

| str | 4 | | m | a | t | . |
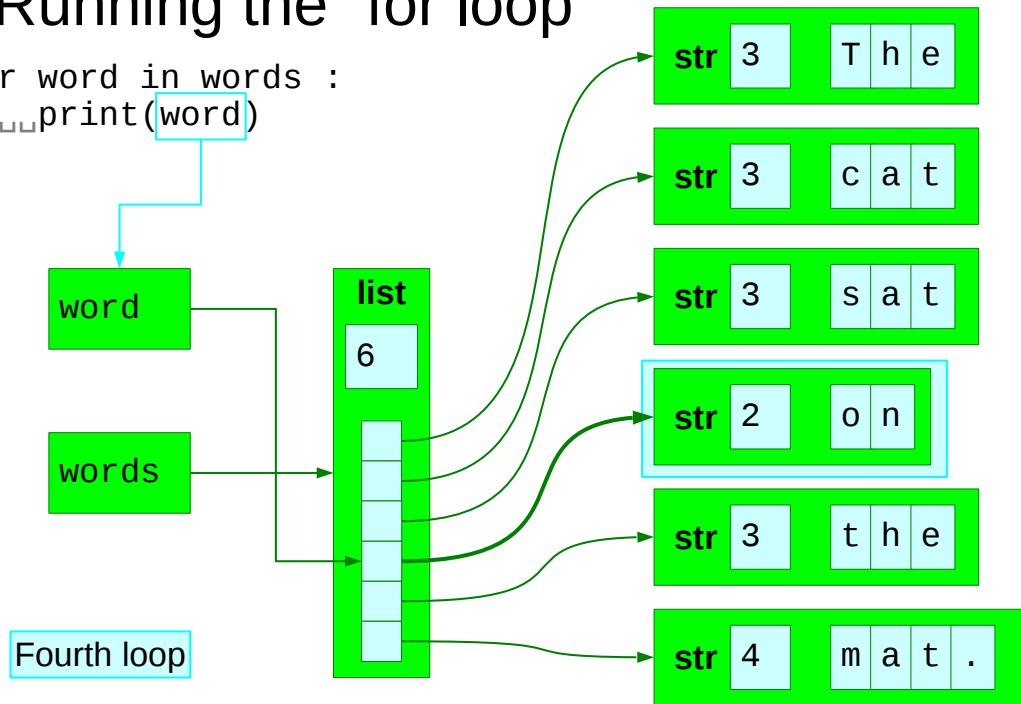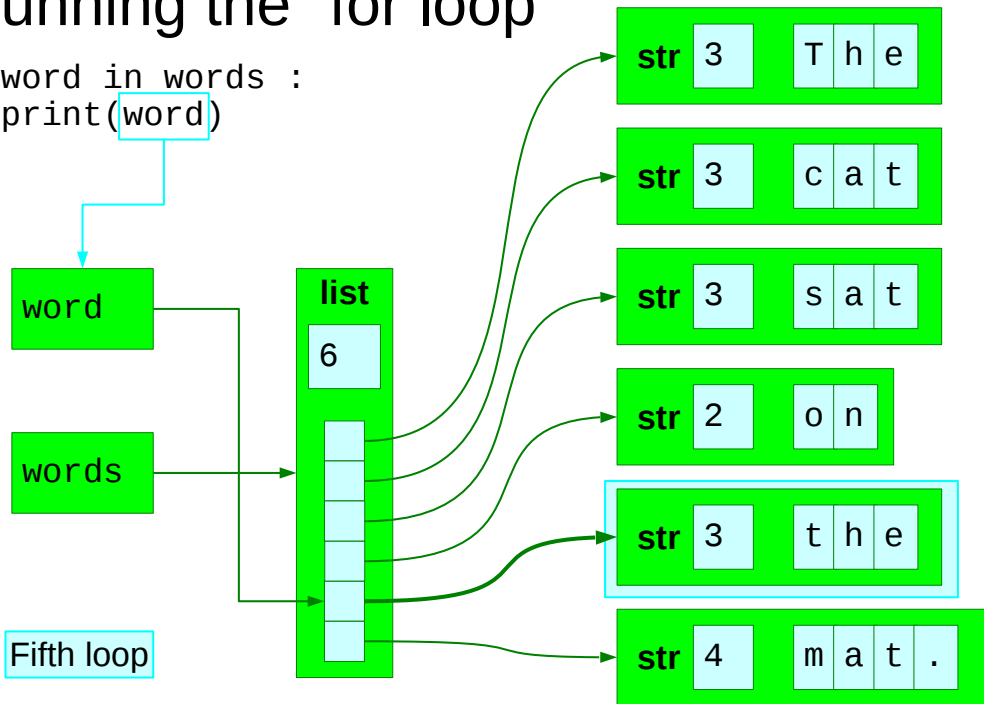
word
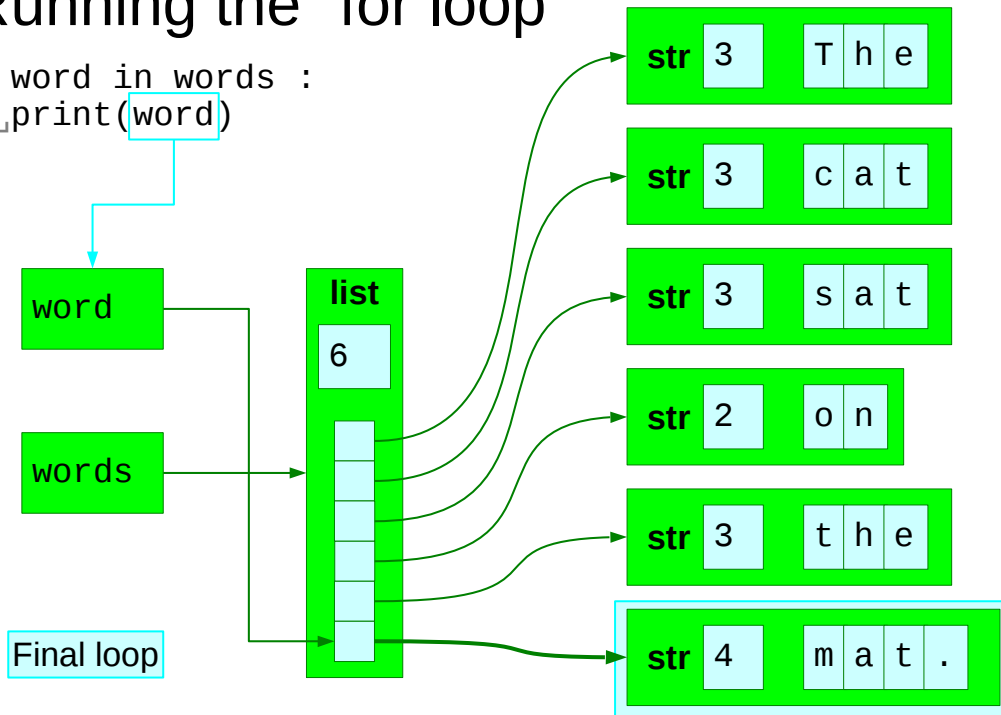
words

Final loop

…until finally it is linked to the final element of the list.
The loop is not run again.

The "for loop" for printing

```
words = ['The', 'cat', 'sat', 'on', 'the', 'mat.']


for word in words :


␣␣␣␣print(word)
```

for1.py

There is a simple example of this in the file `for1.py` in your home directories.

```
$ python3 for1.py
The
cat
sat
on
the
mat.
$
```

The "for loop" for adding

```
numbers = [45, 76, -23, 90, 15]

total = 0                    Set up before the loop


for number in numbers :

    total += number          Processing in the loop

                                 for2.py

print(total)                 Results after the loop
```

261

Our second case was an "accumulator" adding the elements in a list.

Here we establish an initial start value for our total of 0 and give it the name "sum".

Then we loop through the elements in the list, adding their values to the running total as we move through them.

The unindented line after the loop block marks the end of the loop block and is only run after all the looping is completed. This prints out the value of the total now that all the numbers in the list have been added into it.

```
$ python3 for2.py
203
$
```

The "for loop" for creating a new list

```
numbers = [4, 7, -2, 9, 1]

squares = [ ]                          Set up before the loop


for number in numbers :

    squares.append(number**2)   Processing in the loop

                                       for3.py

print(squares)                  Results after the loop
```

Our third example made a new list from the elements of an old list. For example, we might want to take a list of numbers and produce the list of their squares.

In this case the usual process is that rather than have an accumulator with an initial value we start with an empty list and, as we loop through the input values, we append() the corresponding output values.

```
$ python3 for3.py
[16, 49, 4, 81, 1]
$
```

The loop variable persists!

```
numbers = [4, 7, -2, 9, 1]

squares = [ ]

for number in numbers :
    squares.append(number**2)

print(number)
```

Loop variable only meant for use in loop!

But it persists!

263

There is one nicety we should observe.

The loop variable was created for the purpose of running through the elements in the list. But it is just a Python name, no different from the ones we establish by direct assignment. While the `for…` loop creates the name it does not clean it up afterwards.

"for loop hygeine"

```
numbers = [4, 7, -2, 9, 1]

squares = [ ]

for number in numbers :

    squares.append(number**2)

del number
```

number ← Delete it after use

It is good practice to delete the name after we have finished using it. So we will follow our `for…` loops with a del statement.

This is not required by the Python language but we recommend it as good practice.

Progress

Testing items in lists

```
3 in [1,2,3,4] ——► True
```

for loops

```
total = 0
for number in [1,2,3,4]:
    total += number
del number
```

loop variables

```
for number in [1,2,3,4]:
    total += number
del number
```

Exercise 13

```
numbers = [0, 1, 2, 3, 4, 5]

total = 0
total_so_far = []

for number in numbers:
    total += number
    total_so_far.append(total)

print(total_so_far)
```
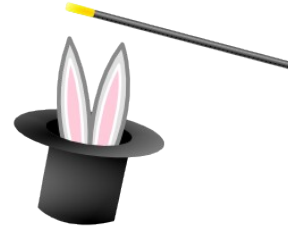
5 minutes

266

"Sort-of-lists"

Python "magic":

Treat it like a list and it will
behave like a *useful* list

What can "it" be?

We have seen already that every Python type comes with a function that attempts to convert other Python objects into that type. So the `list` type has a `list()` function.

However, with lists Python goes further and puts a lot of work into making this transparent and convenient. In very many cases in Python you can drop an object into a list construct and it will act as if it was a list, and a convenient list at that.

Strings as lists

Recall:

```
list('Hello')  ────────►  ['H', 'e', 'l', 'l', 'o']
```

```
for letter in 'Hello' :         ────────►    H
    print(letter)                             e
                                              l
                                              l
  Gets turned                                 o
  into a list.
```

for4.py

268

For example, we know that if we apply the `list()` function to a string we get the list of characters. But the Python "treat it like a list" magic means that if we simply drop a string into the list slot in a `for…` loop then it is treated as exactly that list of characters automatically.

```
$ python for4.py
H
e
l
l
o
$
```

Creating lists of numbers

Built in to Python:

```
range(start,limit)
```

```
for number in range(3,8):
    print(number)
```

⟶ 3
4
5
6
7
☐ ← 8 not included

There are other Python objects which, while not lists exactly, can be treated like lists in a `for…` loop. A very important case is a "`range`" object.

Note that the range defined by 3 and 8 starts at 3 but ends one short. This is part of the whole "count from zero" business.

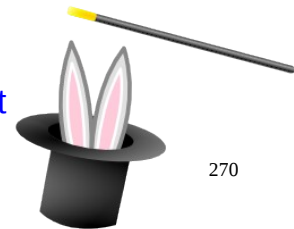# ranges of numbers

Not actually lists:

```
>>> range(0,5)

range(0,5)
```

But close enough:

```
>>> list(range(0,5))

[0, 1, 2, 3, 4]
```

Treat it like a list and
it will behave like a list

270

Strictly speaking a `range` is not a list. But it is close enough to a list that when you drop it into a `for…` loop, which is its most common use by far, then it behaves like the list of numbers.

Why not just a list?

Most common use:

```
for number in range(0, 10000):
    …
```

Inefficient to make a huge list just for this

"iterator"   : anything that can be treated like a list

```
list(iterator)
```
⟶ Explicit list

So why does the `range()` function not just produce a list?

Well, its most common use is in a for… loop. Only one value is required at a time. If the list was explicitly created t would waste computer memory for all the items not in use at the time and computer time for creating them all at once.

(Truth be told, for the purposes of this course you wouldn't notice.)

Ranges of numbers again

via `list()`

```
range(10)        ⟶ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```
Start at 0

```
range(3, 10)     ⟶ [3, 4, 5, 6, 7, 8, 9]
```

```
range(3, 10, 2)  ⟶ [3, 5, 7, 9]
```
Every n^{th} number

```
range(10, 3, -2) ⟶ [10, 8, 6, 4]
```
Negative steps

272

The `range()` function can be used with different numbers of arguments.
A single argument gives a list running from zero to the number.
Two arguments gives the lists we have already seen.
A third argument acts as a "stride" and can be negative.

Indices of lists

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

>>> len(primes)

8

>>> list(range(8))

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

valid indices

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

Now that we have the range object we can move on to one of the most important uses of it.

So far we have used a for… loop to step through the values in a list. From time to time it is important to be able to step through the valid indices of the list.

Observe that if we apply the range() function to a single number which is the length of a list then we get a list of the valid indices for that list.

Direct value or via the index?

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

```
for prime in primes:          Simpler
    print(prime)
```

```
for index in range(len(primes)):    Equivalent
    print(primes[index])
```

274

What good is a list of valid indices?

There are two ways to step through the values in a list. One is directly; this is the method we have met already. The second is to step through the indices and to look up the corresponding value in the list.

These are equivalent and the first method we have already seen is shorter to write. So why bother with the second?

Working with two lists: "dot product"

```
list1 = [0.3,    0.0,    0.4]
    ▪        ×      ×       ×
list2 = [0.2,    0.5,    0.6]


        0.06 + 0.0  + 0.24 ────────▶ 0.3
```

Consider operations on *two* lists.

A concrete example might be the "dot product" of two lists. This is the sum of the products of matching elements.

So

    [0.3, 0.0, 0.4] · [0.2, 0.5, 0.6]

=   0.3×0.2 + 0.0×0.5 + 0.4×0.6

=   0.06 + 0.0 + 0.24

=   0.6

How might we implement this in Python?

Working with two lists: indices

```
              0      1      2
list1 = [0.3, 0.0, 0.4]
list2 = [0.2, 0.5, 0.6]            indices

sum = 0.0

for index in range(len(list1)) :

    sum += list1[index]*list2[index]
                                    Dealing with
print(sum)                          values from
                                    both lists at
                                    the same time.
```

We can approach this problem by running through the valid indices and looking up the corresponding values from each list in the body of the `for…` loop.

A little more about iterators — 1
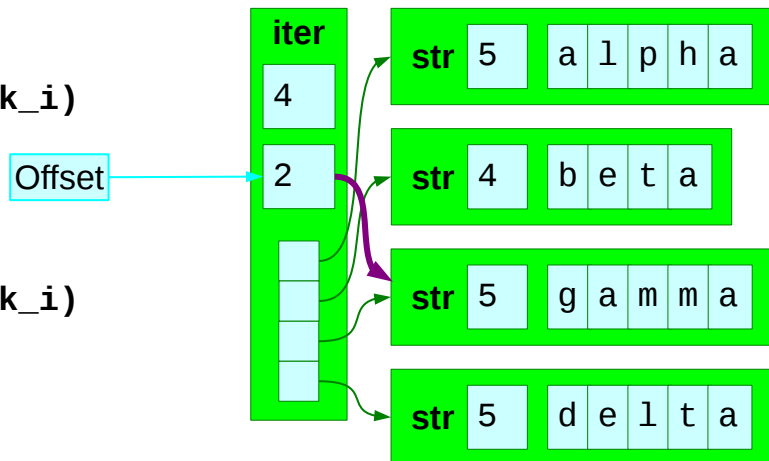
```
>>> greek = ['alpha', 'beta', 'gamma', 'delta']

>>> greek_i = iter(greek)


>>> next(greek_i)

'alpha'


>>> next(greek_i)

'beta'
```

iter

| 4 |
| 2 |
| |
| |
| |

Offset

| **str** | 5 | | a | l | p | h | a |

| **str** | 4 | | b | e | t | a |

| **str** | 5 | | g | a | m | m | a |

| **str** | 5 | | d | e | l | t | a |

We will look a little more closely at iterators so that we can recognise them when we meet them later.

If we start with a list then we can turn it into an iterator with the `iter()` function.

An iterator created from a list is essentially the same as the list with a note of how far through the list we have read. This reference starts at zero, obviously.

Python provides a `next()` function which can act on any iterator which returns the next value (or the first if we've not started) and increments this internal counter.

A little more about iterators — 2
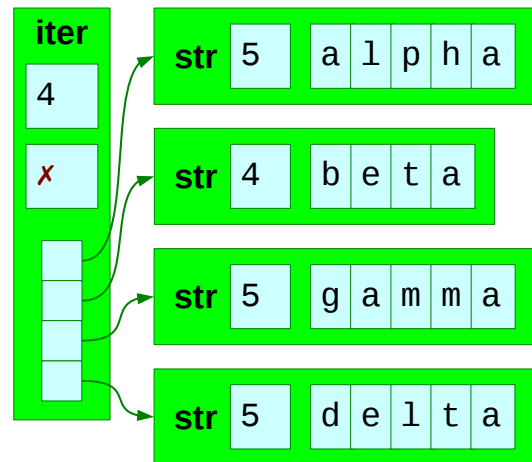
```
>>> next(greek_i)

'gamma'


>>> next(greek_i)

'delta'


>>> next(greek_i)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

iter

4

x

str 5 | a l p h a

str 4 | b e t a

str 5 | g a m m a

str 5 | d e l t a

278

Note that `next()` complains vigorously if we try to run off the end. For this course, where these errors are all fatal it means that we can't use `next()` directly. we don't need to; we have the `for…` loop which handles the error for us.

Progress

Non-lists as lists	```for letter in 'Hello':``` ``` ...```

range()	```range(limit)```
	```range(start,limit)```
	```range(start,limit,step)```
	
	```range(3,7)``` ⟶ ```[3,4,5,6]```

"Iterators"	```greek_i = iter(greek)``` ```next(greek_i)```
Indices of lists	```for index in range(len(things)):```
Parallel lists	

279

Exercise 14

Complete `exercise14.py`

```
list1 = [0.3,   0.0,   0.4]
list2 = [0.2,   0.5,   0.6]
```

            0.1   -0.5   -0.2   Difference

                                        Square

         0.01 + 0.25 + 0.04 ──────────→ 0.3

                                        Add
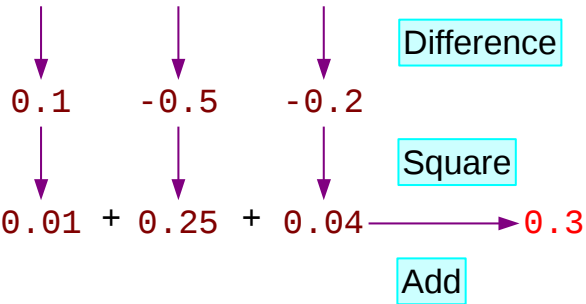
5 minutes

This exercise develops the Python to calculate the square of the distance between two 3D points. We could use our square root Python from earlier to calculate the distance itself but we will meet the "real" square root function later in this course so we'll hold back from that for now.

List "slices"

```
>>> primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

>>> primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]    The list

>>> primes[3]
7                                         An item

>>> primes[3:9]
[7, 11, 13, 17, 19, 23]                   Part of the list
```
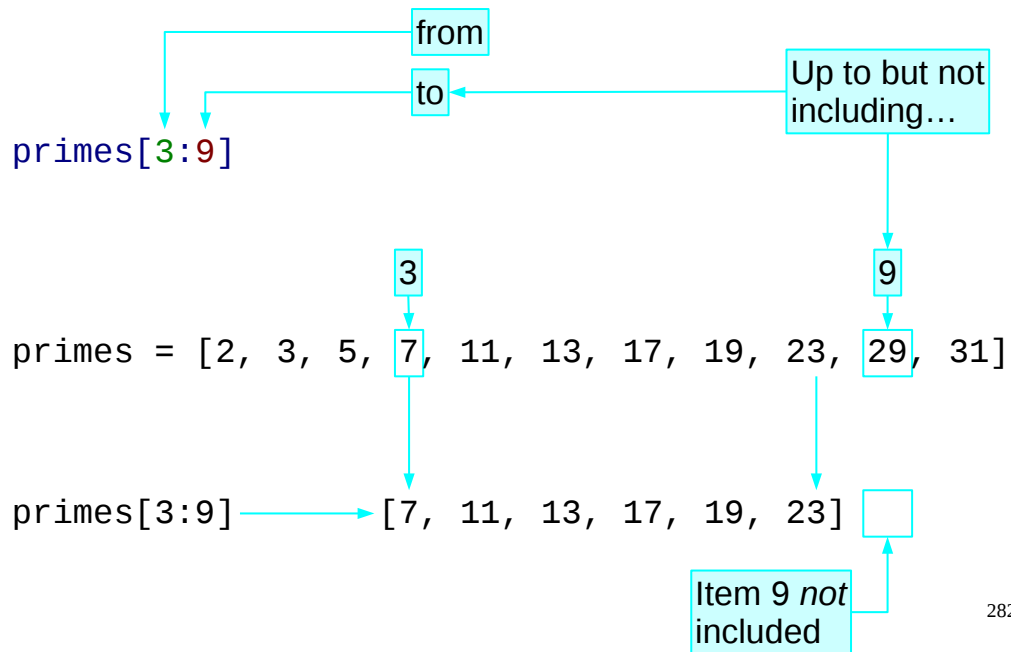
There is one last piece of list Pythonry we need to see.

Python has a syntax for making copies of parts of lists, which it calls "*slices*".

If, instead of a simple index we put two indices separated by a colon then we get the sub-list running from the first index up to but excluding the second index.

Slices — 1

from

to ← Up to but not including…

`primes[`3:9`]`

3

9

`primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]`

`primes[3:9]` ──────► `[7, 11, 13, 17, 19, 23]`

Item 9 *not* included

282

The last index is omitted as part of the "count from zero" thing.

Slices — 2

```
primes         [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]

primes[3:9]         [7, 11, 13, 17, 19, 23]

primes[:9]    [2, 3, 5, 7, 11, 13, 17, 19, 23]

primes[3:]          [7, 11, 13, 17, 19, 23, 29, 31]

primes[:]     [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

283

We can omit either or both of the numbers. Missing the first number means "from the start" and missing the second means "right up to the end".

Slices — 3

```
primes          [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]


primes[3:9]              [7, 11, 13, 17, 19, 23]


primes[3:9:2]            [7,      13,      19    ]


primes[3:9:3]            [7,             17         ]
```

We can also add a second colon which is followed by a stride, just as with `range()`.

Copies and slices — 1

```
>>> letters = ['a','b','c']

>>> alphabet = letters
```
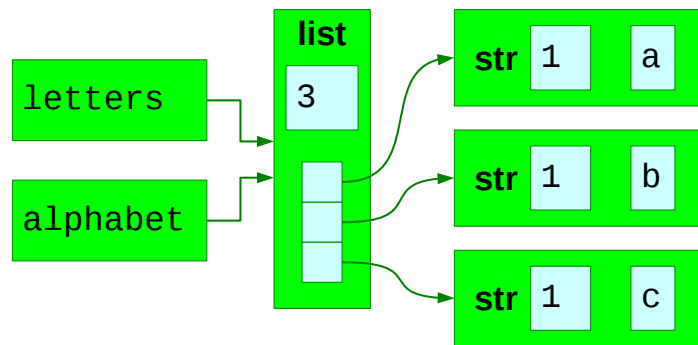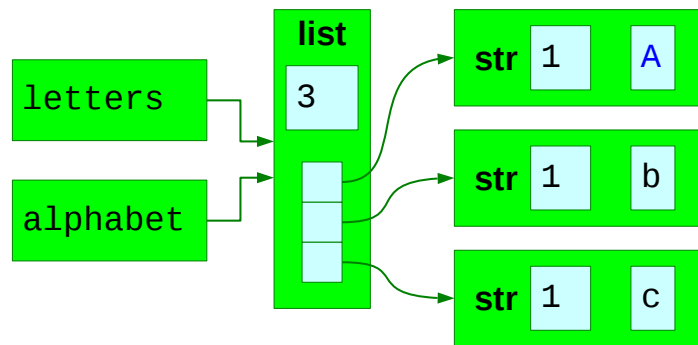

Slices allow us to make copies of entire lists.

If we use simple name attachment then we just get two names for the same list.

Copies and slices — 2

```
>>> letters[0] = 'A'

>>> print(alphabet)

['A', 'b', 'c']
```
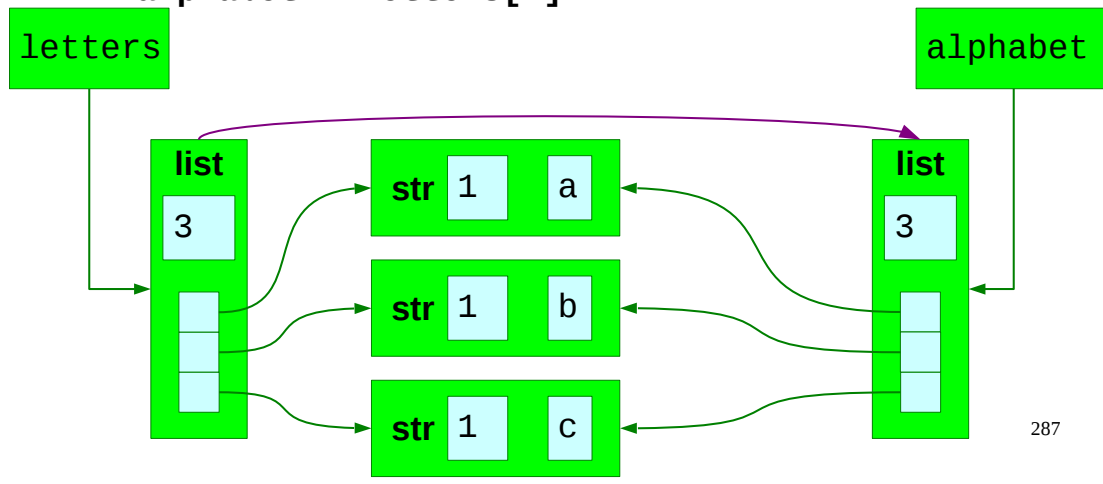
Changing an item in the list via one name shows up via the other name.

Copies and slices — 3

```
>>> letters = ['a','b','c']
```

Slices are copies.

```
>>> alphabet = letters[:]
```

letters

alphabet

list

3

str 1 a

str 1 b

str 1 c

list

3

287

Slices are copies, though, so if we attach a name to a slice from a list — even if that slice is the entire list — then we have two separate lists each with their own name attached.
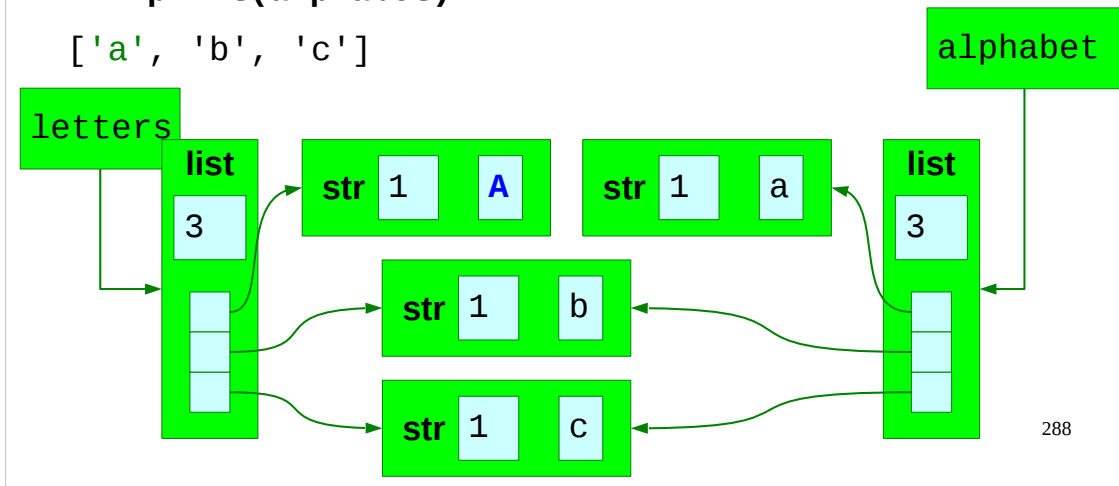
Copies and slices — 4

```
>>> letters[0] = 'A'
```

Slices are copies.

```
>>> print(alphabet)
['a', 'b', 'c']
```

alphabet

letters

| list |
| 3 |

str 1 **A**

str 1 a

str 1 b

str 1 c

| list |
| 3 |

288

So changes in one don't show in the other because they are not the same list.

Progress

Slices

End-limit excluded

Slices are copies

```
items[from:to]          items[from:to:stride]

items[:to]              items[:to:stride]

items[from:]            items[from::stride]

items[:]                items[::stride]
```

Exercise 15

Predict what this Python will do.
Then run it.   exercise15.py
Were you right?

```python
foo = [4, 6, 2, 7, 3, 1, 9, 4, 2, 7, 4, 6, 0, 2]

bar = foo[3:12:3]

bar[2] += foo[4]

foo[0] = bar[1]

print(bar)
```
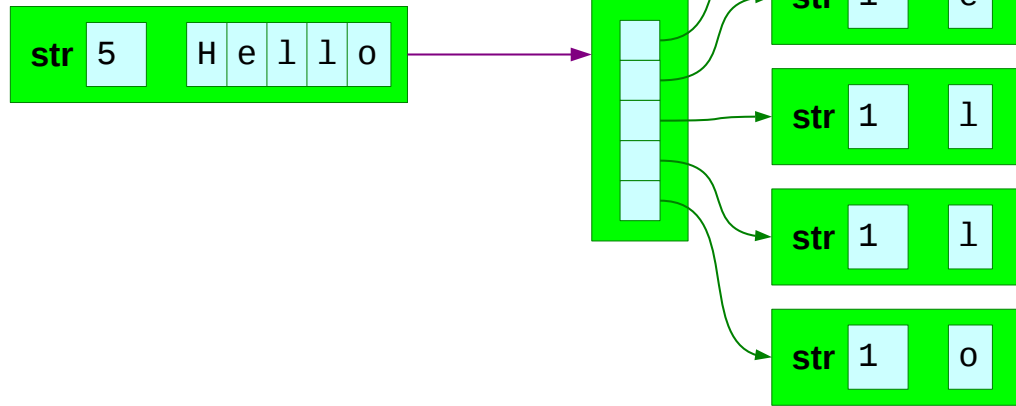
3 minutes

Diversion: Lists and strings

Text: "a **string** of characters"

`list()`

str	5		H	e	l	l	o

list

5

str	1		H

str	1		e

str	1		l

str	1		l

str	1		o

291

Recall that text is called a "string" object because it is a "string of characters". We have already seen that there is a straightforward way, using the `list()` function, to turn a string into the literal list of characters.

Indexing into strings

```
>>> 'Hello, world!'[0]
'H'
```
Simple indexing

```
>>> 'Hello, world!'[:4]
'Hell'
```
Slicing

We can go further. It is actually possible to refer to single characters out of strings by giving the position of that character as an index.

We can create substrings of the text by using slicing. Note that this gives a second string, not a list.

Strings are immutable

```
>>> 'Hello, world!'[0] = 'C'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support
            item assignment
```
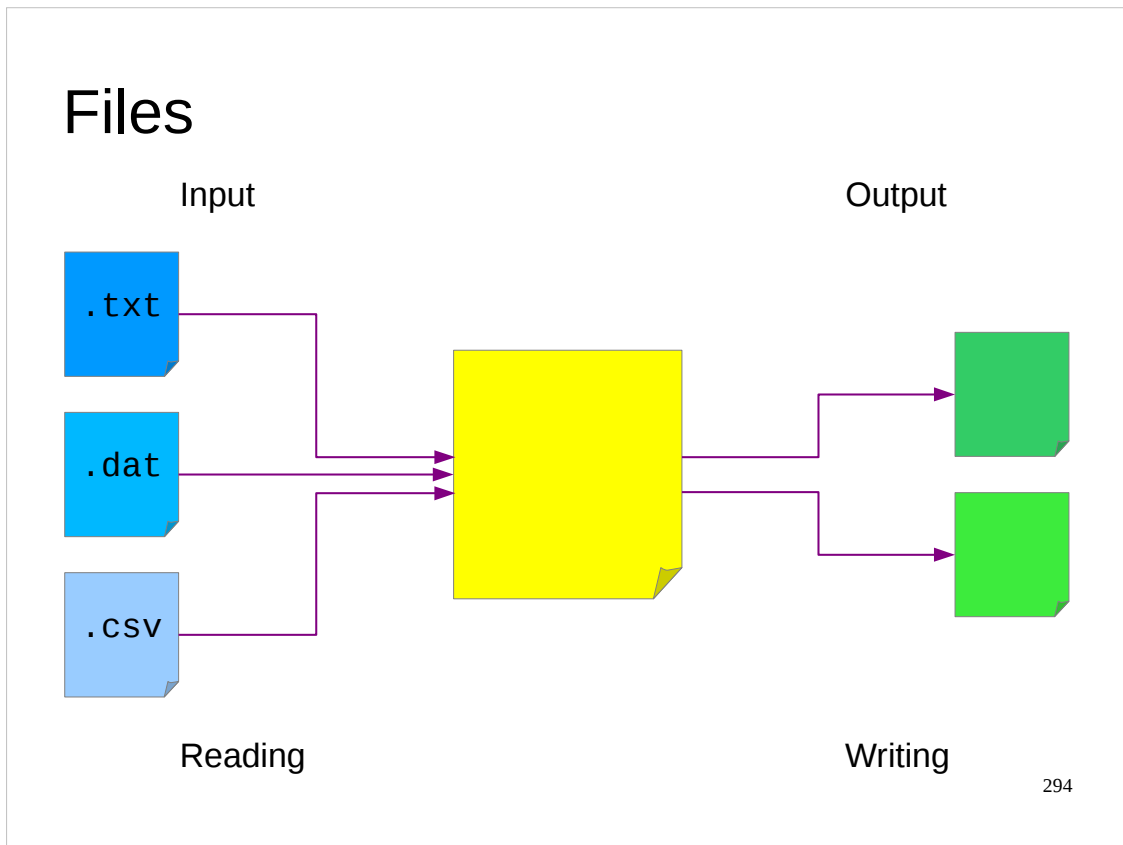
However, we cannot use the index notation on the left hand side of an assignment.

We cannot use this to change the contents of a string. String objects in Python are immutable, just like integers. If you want a different string you have to create it from scratch.
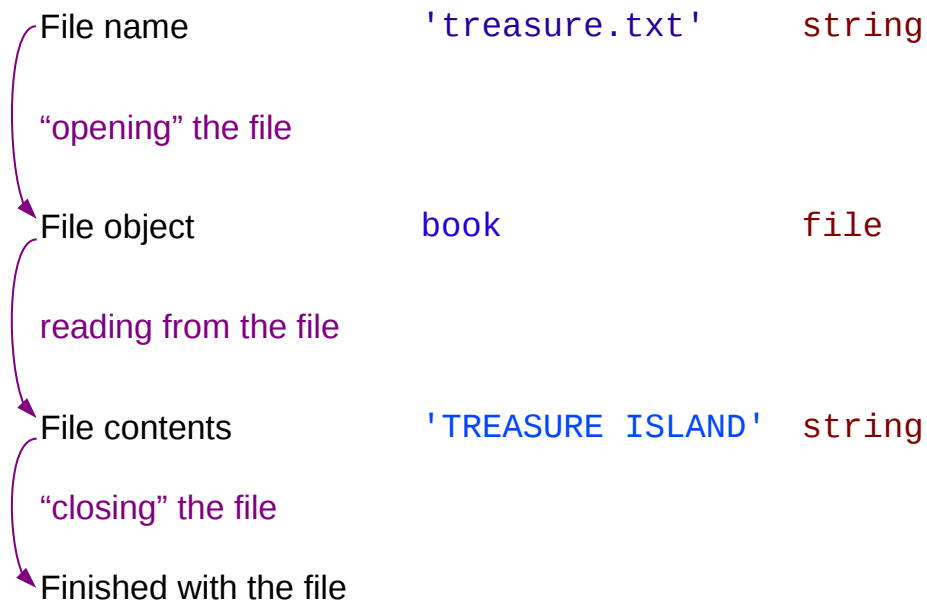
Files

Input

.txt

.dat

.csv

Reading

Output

Writing

294

Now we will look at something completely different that will turn out to be just like a list: Files.

We want our Python scripts to be able to read in and write out files of text or data.

We will consider reading files first and writing them second.

Reading a text file

File name   `'treasure.txt'`   `string`

"opening" the file

File object   `book`   `file`

reading from the file

File contents   `'TREASURE ISLAND'`   `string`

"closing" the file

Finished with the file

Reading from a file involves four operations bracketing three phases.

We start with a file name. This is a string of characters.

I want to be pedantic about something in this course: a file *name* is not a file. A file is a lump of data in the computer's long-term store. A file name is a short piece of text.

We link a file name to a file by a process called "opening the file". This takes the file name and creates a Python file object which will act as our conduit to the file proper.

We will use this file object to read data from the file into the Python script.

When we are done reading data out of the file (via the file object) we will signal to both python and the operating system that we are done with it by "closing" it. This disconnects us from the file and we would have to re-open it if we wanted more data.

Opening a text file

Python function

File name

Read-only [text]

>>> **book = open('treasure.txt', 'r')** "mode"

Python file object

Read-only is the default

        book = open('treasure.txt'   )
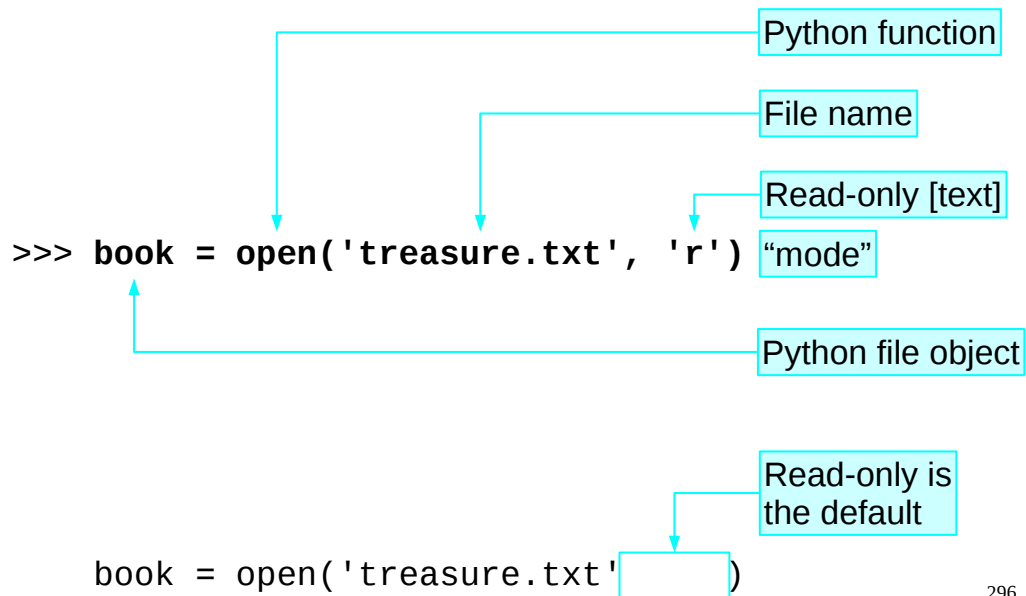
We will start with opening a file.

We start with just the file name. This is passed into the open() function with a second argument, `'r'`, indicating that we only want to **r**ead the file.

The function hooks into the operating system, which looks up the file by name, checks that we have permission to read it, records the fact that we are reading it, and hands us back a handle — the file object — by which we can access its contents.

Reading from a file object

"next line, please"

File object

```
>>> line1 = next(book)
```

First line of file

```
>>> line1
```

Includes the "end of line"

```
'TREASURE ISLAND\n'
```

Now that we have this hook into the file itself, how do we read the data from it?

File objects are iterators, so we can apply the next() function to them to get the next line, starting with the first.

File object are iterable

```
>>> line2 = next(book)
```
Second line of file

```
>>> line2
'\n'
```
A blank line

```
>>> line3 = next(book)
```

```
>>> line3
```
Third line of file
```
'PART ONE\n'
```

Note that a "blank" line actually contains the end of line character. It has length 1, and is not an empty string, length 0.

Closing the file

Method built in to file object

```
>>> book.close()
```

Frees the file for other
programs to write to it.

When we are done reading the data we need to signal that we are done with it. Python file objects have a `close()` method built into them which does precisely this.

The file object — 1

```
>>> book

<_io.TextIOWrapper
  name='treasure.txt'
  encoding='UTF-8'>
```

Text Input/Output

File name

Character encoding: how to represent letters as numbers.

300

UTF-8 means "UCS Transformation Format — 8-bit".

UCS means ""Universal Character Set. This is defined by International Standard ISO/IEC 10646, Information technology — Universal multiple-octet coded character set (UCS). For more information than you could possibly want on this topic visit the Unicode web pages: `www.unicode.org`.

The file object — 2
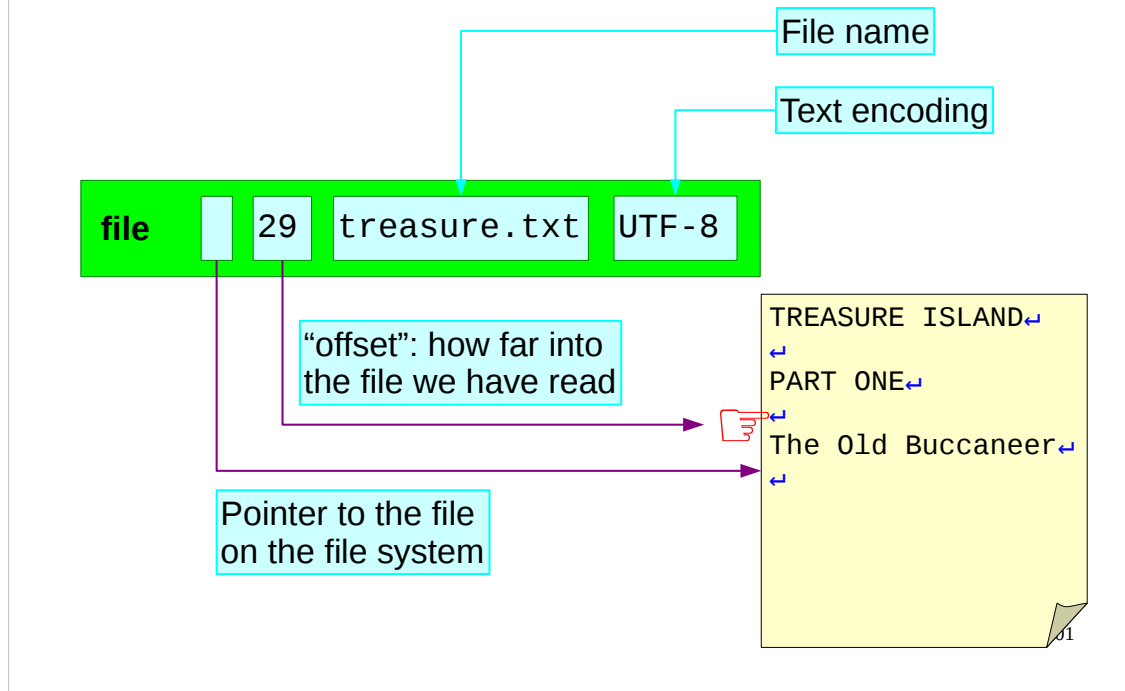
The Python file object contains a lot of different bits of information. There are also lots of different sorts of file object, but we are glossing over that in this introductory course.

What they share is a reference into the operating system's file system that identifies the specific file and acts as the declaration to the operating system that the Python process is using the file.

They also share an "offset". This is a number that identifies how far into the file the program has read so far. The `next()` function reads from the offset to the next new line and increases the offset to be the distance into the file of the new line character.

The file object also records the name it was opened from and the text encoding it is using to convert bytes in the file to characters in the program.

Reading through a file

Treat it like a list and
it will behave like a list

list(*file_object*) ─────────► List of lines in the file

```
>>> book = open('treasure.txt', 'r')

>>> lines = list(book)

>>> print(lines)
```

302

Given that a file is an iterator, we can simply convert it into a list and we get the list of lines.

Reading a file moves the offset ⚠

```
>>> book = open('treasure.txt', 'r')


>>> lines_a = list(book)

>>> print(lines_a)

...          ⟵          Huge output

>>> lines_b = list(book)

>>> print(lines_b)

[]          ⟵          Empty list
```

303

Note, however, that the act of reading the file to get the lines reads through the file, so doing it twice gives an empty result second time round.

Reading a file moves the offset

```
>>> book = open('treasure.txt', 'r')
```
File object starts with offset at **start**.

```
>>> lines_a = list(book)
```

```
>>> print(lines_a)
```
Operation reads entire file from offset.

Offset changed to **end** of file.

```
...
```

```
>>> lines_b = list(book)
```

```
>>> print(lines_b)
```
Operation reads entire file from offset.

So there's nothing left to read.

```
[]
```
304

recall that the reading starts at the offset and reads forwards — to the end of the file in this example. It also moves the offset to what was last read. So the offset is changed to refer to the end of the file.

Resetting the offset

```
>>> book = open('treasure.txt', 'r')

>>> lines_a = list(book)

>>> print(lines_a)

…

>>> book.seek(0)

>>> lines_b = list(book)

>>> print(lines_b)
```

Set the offset explicitly

We can deliberately change the offset. For text files with lines of different lengths this can be more complex that you would imagine. The only safe value to change the offset to is zero which takes us back to the start of the file.

Typical way to process a file

```
book = open('treasure.txt', 'r')


for line in book :          Treat it like a list…

    …
```

Being able to read a single line of a file is all very well. Converting an entire book into a list of its lines might prove to be unwieldy.

What is the typical way to do it?

Given that files can be treated like lists the easiest way to process each line of a file is with a `for…` loop.

Example: lines in a file

```
book = open('treasure.txt', 'r')


n_lines = 0          Line count


for line in book :   Read each line of the file

    n_lines += 1     Increment the count


print(n_lines)       Print out the count


book.close()
```

307

For example, we could just increment a counter for each line to count the number of lines.

Example: characters in a file

```
book = open('treasure.txt', 'r')


n_chars = 0


for line in book :

    n_chars += len(line)


print(n_chars)


book.close()
```

Number of characters on the line

Increase the count by that many

We could measure the length of the line and increment the counter by the number of characters in the line to count the total number of characters in the file.

Progress

Opening file to read

```
book = open(filename, 'r')
```

Reading file

```
for line in book:
    ...
```

Closing file

```
book.close()
```

File offset

```
book.seek(0)
```

309

Exercise 16

Complete a script to count
lines, words and characters
of a file.

`exercise16.py`

5 minutes

Hint: recall the `split()` method that strings have:
```
>>> 'Hello, world!'.split()
['Hello,', 'world!']
```

Once this script is complete you will have written an equivalent of the standard (albeit simple) Unix command line tool "**wc**":
```
$ python3 exercise16.py
File name? treasure.txt
treasure.txt 8734 67945 364589

$ wc treasure.txt
  8734  67945 364589 treasure.txt
```

Writing files

File name   `'treasure.txt'` `string`

"opening" the file

Data to write   `'TREASURE ISLAND'` `string`

writing to the file

File object   `book`   `file`

"closing" the file

Finished with the file
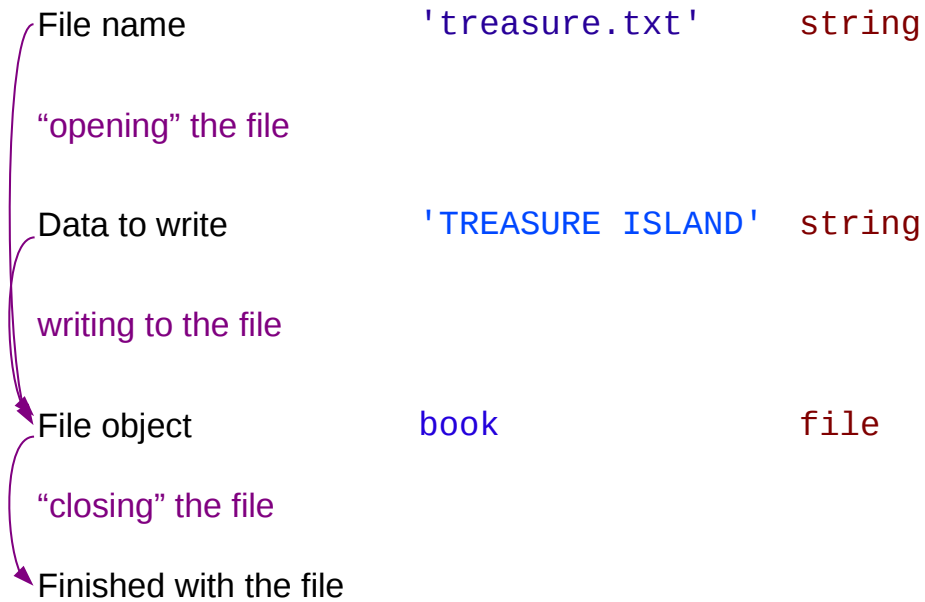
Enough of reading files Robert Louis Stephenson has prepared for us. Let's write our own.

This is again a three phase process. We will open a file for writing, write to it and then close it.

Opening a text file for writing

Open for
writing [text]

```
>>> output = open('output.txt', 'w')
```

This will *truncate*
an existing file

Opening a file for writing is the same as opening it for reading except that the mode is `'w'` for **w**riting.

If the file already exists then this overwrites it. The file gets truncated to zero bytes long because you haven't written to it yet.

Opening a text file for *appending*

Open for
appending [text]

```
>>> output = open('output.txt', 'a')
```

If you do not want to overwrite the content of the file, you can also open a file for appending with mode `'a'`.

In all other regards appending will be the same as writing.

Writing to a file object — 1

File object

Method built in
to file object

```
>>> output.write(line1)

6
```

Data being written

Number of
characters
actually written

```
>>> len(line1)

6
```
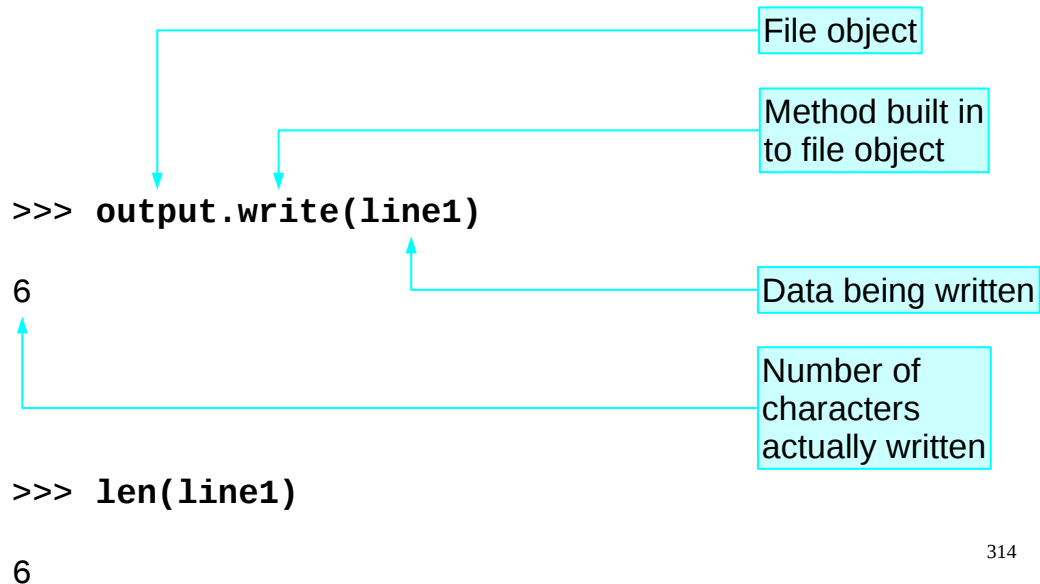
A writeable file object has a method write() which takes a text string (typically but not necessarily a line) and writes it to the file.

This has a return value, which is the number of characters it managed to write. Note that if there are "issues" (running out of space, malfunctions, network interruptions) the system may not be able to write all the characters. Typically it does, though, and the return code should match the length of the line.

Writing to a file object — 2

```
>>> output.write('alpha\n')
6
```
Typical use: whole line.

Includes "end of line"

```
>>> output.write('be')
2
```
Doesn't have
to be whole lines

```
>>> output.write('ta\n')
3
```

```
>>> output.write('gamma\ndelta\n')
12
```
Can be multiple lines

How the data is chopped up between writes is up to you.

Closing the file object

```
>>> output.close()
```
Vital for written files

It also has a `close()` method.

Importance of closing

!

Data "flushed" to disc on closure.

write   "flush"

Python script   File object   File system   317
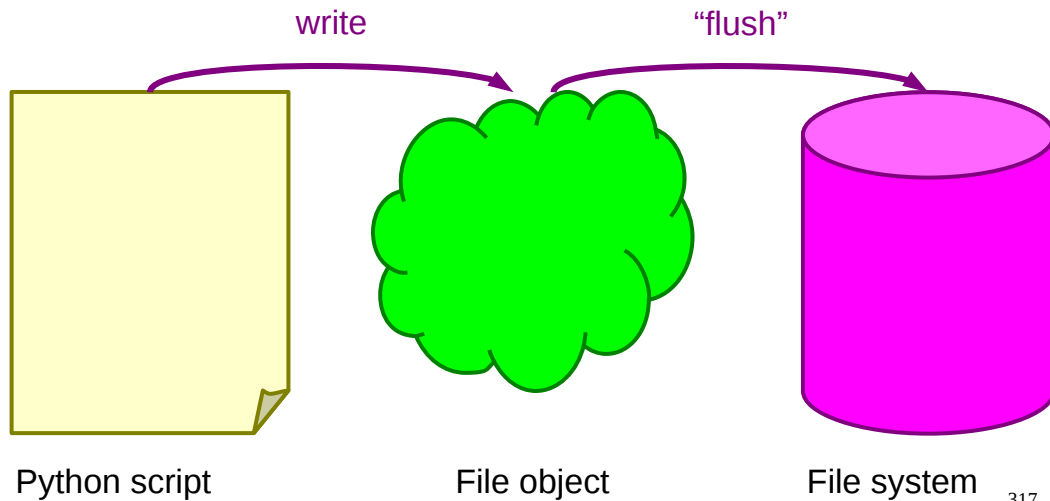
Closing files is even more important for written files that read ones.

It is only when a file is closed that the data you have written to the Python file object is definitely sent to the operating system's file. This is a process called "flushing the file to disc".

Writing to disc is slow by computing standards so, for efficiency reasons, systems like Python tend to wait until they have at least a certain amount of data for a file before they flush it to disc. If your program exists before flushing then the data may be lost. Closing a file signals that there will be no more data coming for it so Python flushes whatever it has in hand to the disc as part of the closure process.

Importance of closing *promptly*

Files locked for other access

open ['**w**']　　　✗　　　open ['**r**']

318
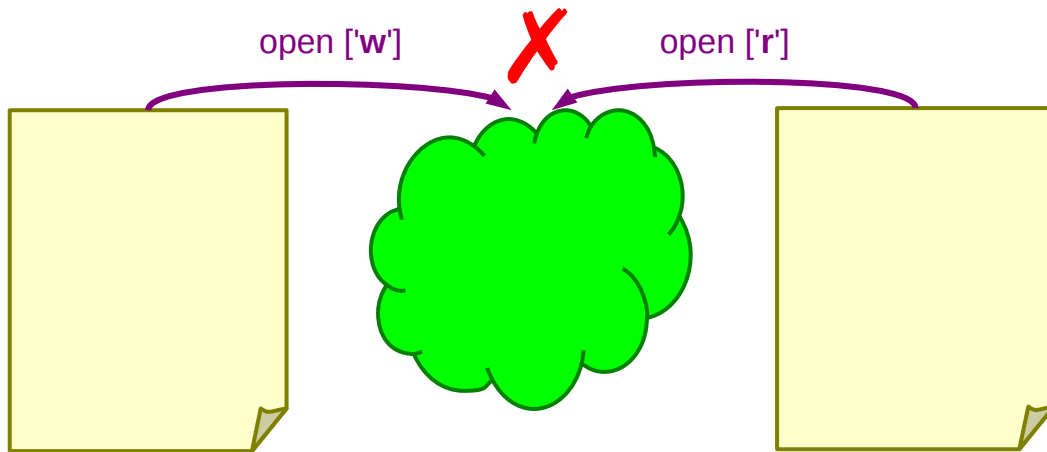
(More a problem for Windows than Unix)

There's more to it than just flushing, though. Holding a file open signals to the underlying computer operating system that you have an interest in the file. How the operating system reacts to this varies from system to system, but Microsoft Windows™ will lock a file so that if you have it open for writing nobody else can open it for reading, even if you don't plan to write any more to it than you have already.

Writing non-text values

```
>>> output.write('Boo!\n')
```
Writing text (str)

```
5
```

```
>>> output.write(42)
```
Writing non-text (int)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

write() only accepts text
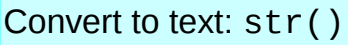
There is one last issue to address with writing data to files.

IUnlike the `print()` function, the `write()` function can only handle text arguments. The `print()` function automatically converts non-text values; `write()` does not. Therefore, we must do the conversion ourselves.

Writing non-text values
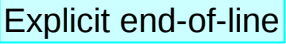
```
>>> output.write(str(42))
2

>>> output.write('\n')
1
```

Convert to text: `str()`

Explicit end-of-line

Text formatting (later in the course)
provides a more elegant solution.

320

We do this simply by calling the `str()` function which converts (almost) anything directly into text.

We also need to explicitly add on the new line character to indicate the end of the line with the data on it.

Later in this course we will see a system Python has for generating formatted text. This will provide a more elegant solution but, until then, we will make do with `str()` and \n.

Progress

Opening files for writing

```
book = open(filename, 'w')
```

Writing text

```
book.write(text)
```

Writing data

```
book.write(str(data))
```

Explicit ends of lines

```
book.write('\n')
```

Closing the file is *important*

```
book.close()
```

Exercise 17

The script `exercise17.py` prints
a series of numbers ending in 1.

Change the script to write to a file called
`output.txt` instead of printing to the console.

5 minutes

322

Functions

$$y = f(x)$$

Functions we have met

```
input(prompt)              bool(thing)

len(thing)                 float(thing)

open(filename, mode)       int(thing)

print(line)                iter(list)

type(thing)                list(thing)

ord(char)                  range(from, to, stride)

chr(number)                str(thing)
```

Not that many!

"The Python Way":
If it is appropriate to an object,
make it a method of that object.

This is the complete set of Python functions that we have met to date.

Actually it's surprising how *few* there are, not how many. Python's philosophy leads to functions that only make sense for a particular sort of object being methods of that object, not free-standing functions.

We are now going to write our own functions.

Why write our own functions?

Easier to …

… read

… write

… test

… fix

… improve

… add to

… develop

"Structured programming"

Why?

Moving our scripts' functionality into functions and then calling those functions is going to make *everything* better. This is the first step towards "structured programming" which is where programming goes when your scripts get too long to write in one go without really thinking about it.

Defining a function

$$(y_1, y_2, y_3) = f(x_1, x_2, x_3, x_4, x_5)$$

Identify the inputs

Identify the processing

Identify the outputs

326

So what do we need to do?

Well any functions starts by defining what inputs it needs, what it does with those inputs to generate the results and exactly what results/outputs it generates.

A function to define: `total()`

Sum a list

[1, 2, 3] ──────────▶ 6

[7, -4, 1, 6, 0] ───────▶ 10

[ ] ──────────▶ 0   "Edge case"

To give ourselves a simple but concrete example to keep in mind we will set ourselves the challenge of writing a function that sums the elements of a list.

This may sound trivial but immediately raises some interesting cases that need to be considered.

What is the sum of an empty list? Zero? Is that an integer zero or a floating point zero? (Or a complex zero?)

We will say it should sum to an integer zero.

Defining a Python function — 1

define a function called …

name of function

inputs

`def` `total`(…): colon
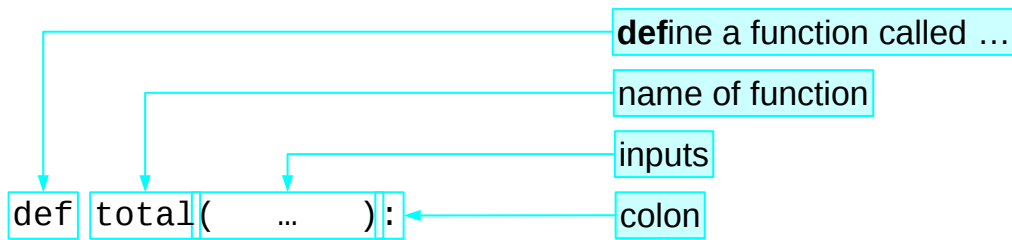
We will plunge straight into the Python.

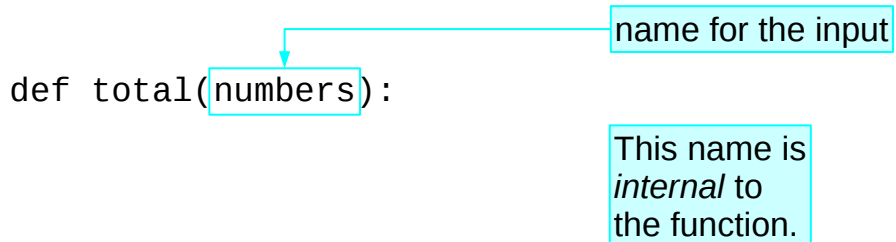The Python keyword to **def**ine a function is "`def`".

This is followed by the name of the function, "`total`" in this case.

This is followed by a pair of round brackets which will contain all the input values for the function.

Finally there is a colon to mark the end of the line and the beginning of the body of the function.

Defining a Python function — 2

name for the input

```
def total(numbers):
```

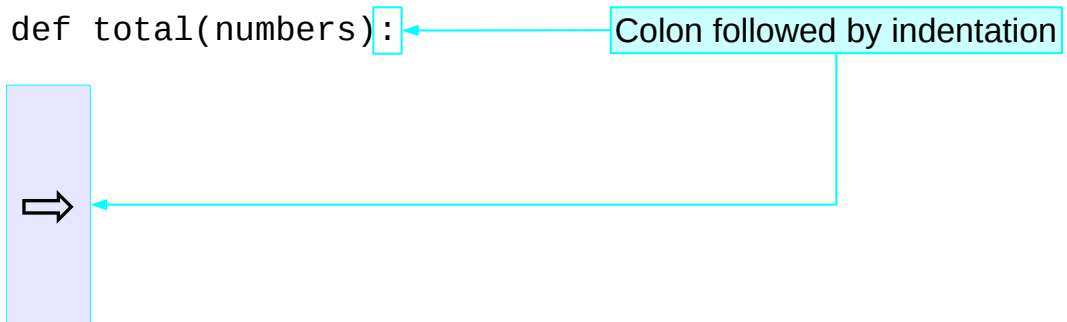This name is *internal* to the function.

Our function takes a single input: the list of numbers to be summed.

What goes inside the brackets on the `def` line is the name that this list will have inside the function's definition. This is the "*x*" in maths. This internal names is typically unrelated to the name the list has in the main body of the script.

It is always a good idea to name your inputs (and other variables) meaningfully. Please try to avoid calling them "*x*", "*y*", or "*z*".

We will call ours "`numbers`".

Defining a Python function — 3

```
def total(numbers): 
```
Colon followed by indentation

As ever with Python a colon at the end of a line is followed by an indented block of code. This will be the body of the function where we write the Python that defines what the function actually does with the input(s) it is given.
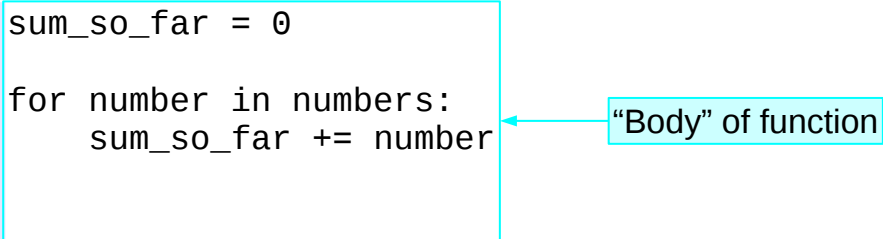
Defining a Python function — 4

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number
```

"Body" of function

For our function this is particularly simple.

Defining a Python function — 4

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number
```

These variables exist *only* within the function's body.

The `numbers` name we specified on the `def` line is visible to Python only within the function definition. Similarly any names that get created within the function body exist only within that function body and will not be visible outside. Nor will they clash with any other uses of those names outside the function body.

In our example code the name "`numbers`" is defined in the `def` line, the "`sum_so_far`" name is defined explicitly in the function body and the "`number`" name is defined by the `for…` loop as its loop variable.

None of these interact with the Python outside the function definition.

Defining a Python function — 5

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far
```

This value is returned

return this value

Finally we need to specify exactly what value the function is going to return. We do this with another Python keyword, "`return`". The value that follows the `return` keyword is the value returned by the function.

When Python reaches the `return` statement in a function definition it hands back the value and ends the execution of the function body itself.

Defining a Python function — 6

And that's it!

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far
```

Unindented after this

And that's all that is involved in the creation of a Python function.

Defining a Python function — 7

And that's it!

All internal names *internal* ———► No need to avoid reusing names

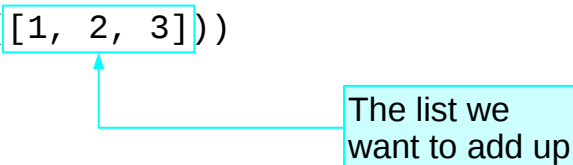All internal names cleaned up ——► No need for `del`

Note that because of this isolation of names we don't have to worry about not using names that are used elsewhere in the script.

Also, as part of this isolation all these function-internal names are automatically cleared when the function finishes. We do not need to worry about deleting them.

Using a Python function — 1

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far


print(total([1, 2, 3]))
```
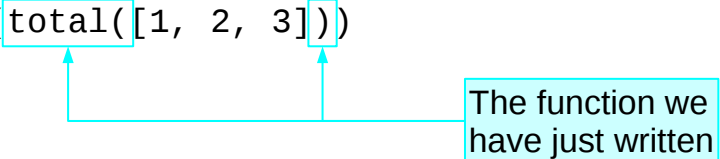
The list we want to add up

336

We use this function we have defined in exactly the same way as we would use a function provided by Python.

Using a Python function — 2

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far


print(total([1, 2, 3]))
```

The function we have just written

337

Using a Python function — 3

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far


print(total([1, 2, 3]))
```

Printing out the answer

338

Using a Python function — 4

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far



print(total([1, 2, 3]))
```
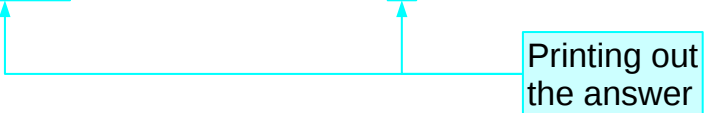
total1.py

nb: Unix prompt

$ **python3 total1.py**

6

The file total1.py in your home directories contains exactly the code you see here.

Using a Python function — 5

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far



print(total([1, 2, 3]))
print(total([7,-4,1,6,0]))
print(total([]))
```

total2.py

```
$ python3 total2.py

6
10
0
```

Use the function
multiple times

Functions' private names — 1

```
def total(numbers):

    sum_so_far = 0

    for number in numbers:
        sum_so_far += number

    return sum_so_far


data = [1, 2, 3]
data_sum = total(data)
print(data_sum)
```

Function definition

Main script

341

We will revisit the idea of names being private to functions and illustrate it with a simple example.

Suppose we had a script that defined the `total()` function and had three lines of the main program as shown.

Functions' private names — 2

```
def total(numbers):
    ...
```

main script

total → function ⚙⚙⚙⚙

"namespace"
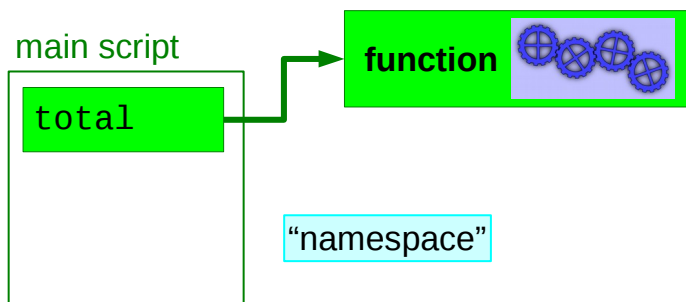
As Python interprets the function it starts by adding a name `total` to the set of names the script knows about and attaches that name to wherever the script's definition is in memory. The name for this collection of names is the script's "namespace".

Functions' private names — 3

```
data = [1, 2, 3]
```
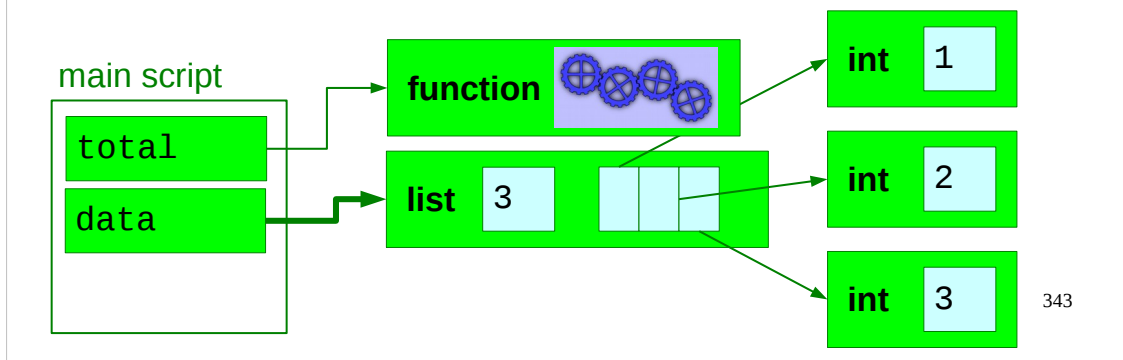

Then Python interprets the first of the following three lines. This establishes a list of three integers and attaches a name to it in the script's namespace.

Functions' private names — 4

```
... = total(data)                    def total(numbers):
```

total

numbers

main script

total

data

function

list 3

int 1

int 2

int 3

344
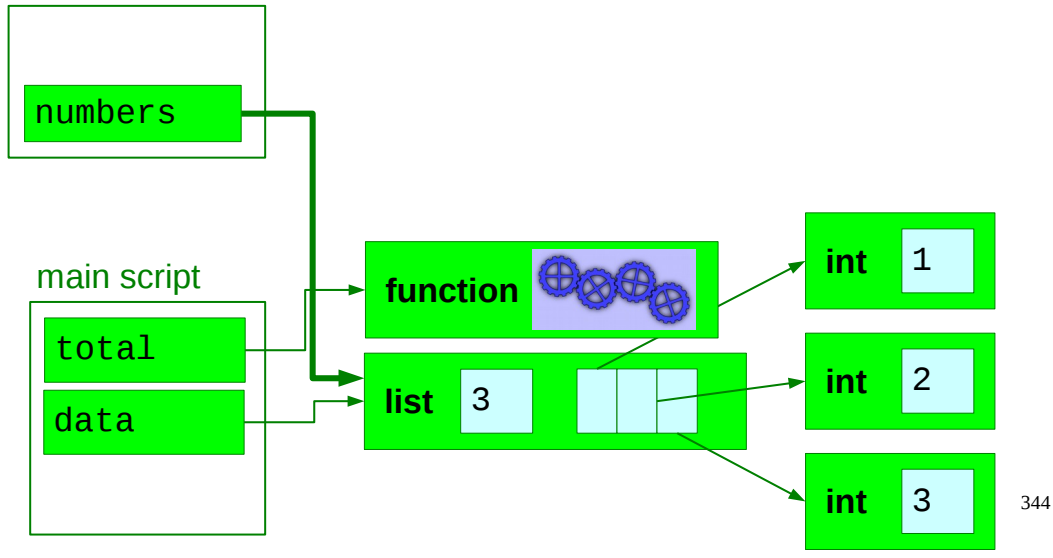
The next line calls the function. As ever with assignment lines we start with the right hand side. The RHS calls the function `total` with argument `data`.

When we call the function a separate namespace is created for it. Within the definition of the function the name "numbers" is used for the list. As a result, the name "`numbers`" is established in the *function's* namespace. This name is attached to the same Python object as the name "`data`" in the *script's* namespace.

Functions' private names — 5

```
... = total(data)                    sum_so_far = 0
```

total

```
sum_so_far
```

int 0

```
numbers
```

main script

```
total
```

function

```
data
```

list 3

int 1

int 2

int 3   345

Next, Python starts to execute the function's body.

The first line in the body defines a name `sum_so_far` and attaches it to a value `0` (integer). Because this name appears in the function body it is established in the function's namespace.

Functions' private names — 6

```
... = total(data)                    return sum_so_far
```

total

sum_so_far → int   **6**

numbers

main script

total

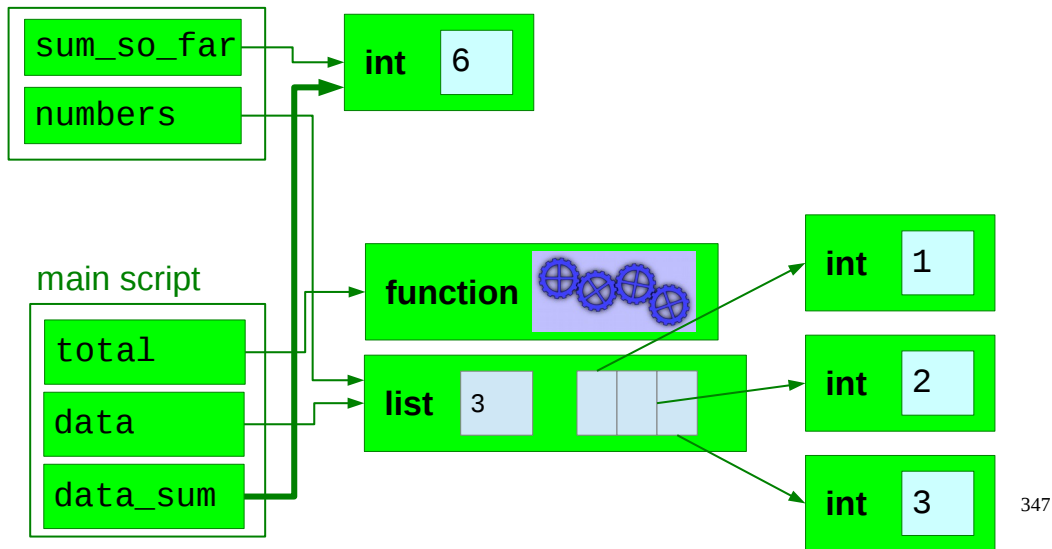data

function

list   3

int   1

int   2

int   3   346
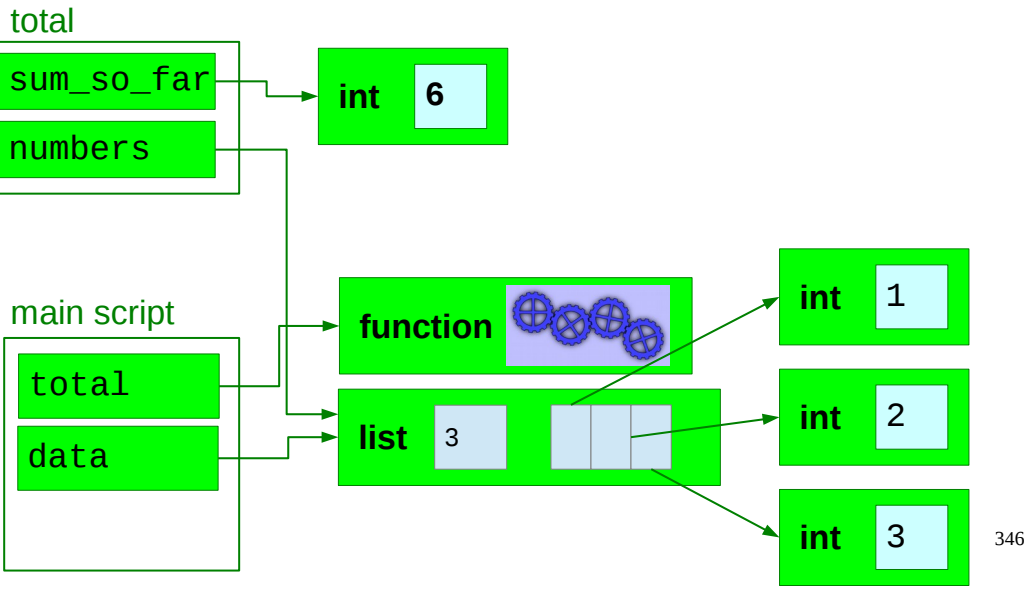
By the end of the script's execution the name `sum_so_far` in the script's namespace is attached to the value 6 (integer).

It is this *value* that is returned.

Functions' private names — 7

```
data_sum = total(data)          return sum_so_far
```

total

sum_so_far

numbers

int   6

main script

total

data

data_sum

function

list   3

int   1

int   2

int   3   347

We now turn to the left hand side of the line in the main script.

This establishes a name `data_sum` to attach to the value returned from the function call. The name is defined in the main body fo the script and so is created in the script's namespace.

The name is attached to the value 6 (integer) that was created by the function.

Functions' private names — 8

```
data_sum = total(data)
```
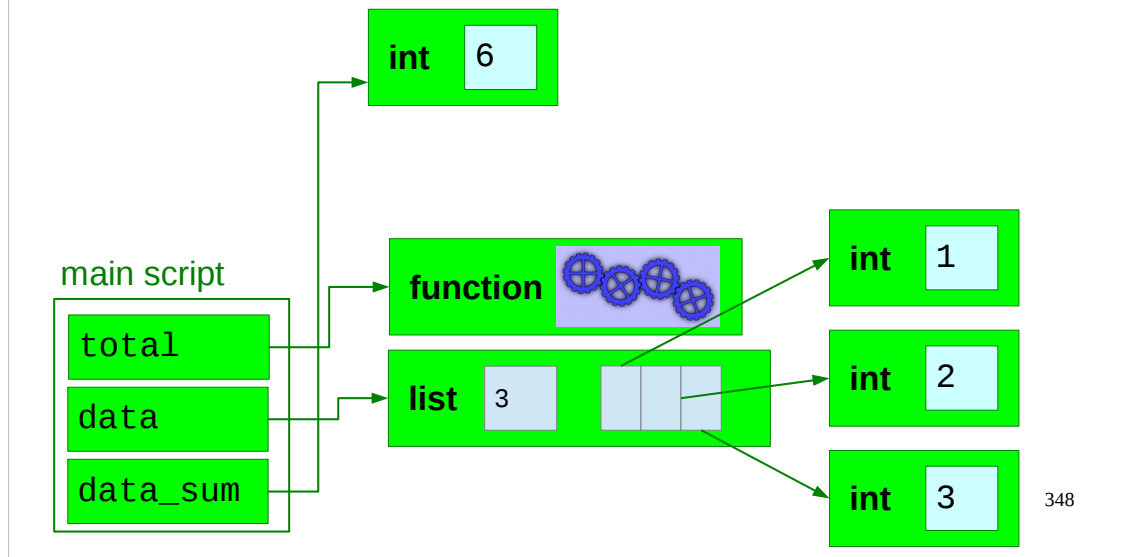

Now that Python is done with the function garbage collection moves in and erases all trace of it. Its namespace is deleted, leaving only the names defined in the main script's namespace.

Progress

Functions

"Structured programming"

Defining a function   `def function(input):`
                                            `...`

Returning a value   `return output`

Private name spaces

349

Exercise 18

Edit the script `exercise18.py`.

It currently defines and uses a function `total()` which adds
the elements of a list.
Change it to create a function `product()` which multiplies them.

Three examples are included in the script.
Running it should produce the following output:

```
$ python3 exercise18.py
6
0
1
```

5 minutes

350

Reminder about indices

```python
def total(numbers):
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number
    return sum_so_far
```

total2.py

Equivalent

```python
def total(numbers):
    sum_so_far = 0
    for index in range(len(numbers)):
        sum_so_far += numbers[index]
    return sum_so_far
```

total3.py

351

Let's quickly remind ourselves about how we can uses indices for lists rather than values from lists directly.

We found this particularly useful when we were traversing more than one list at once.

Example of multiple inputs

Want a function to add two lists of the same length term-by-term:

[1, 2, 3]   &   [5, 7, 4] ⟶ [6, 9, 7]

[10, -5]   &   [15, 14] ⟶ [25, 9]

[3, 7, 4, 1, 7]   &   [8, 4, -6, 1, 0] ⟶ [11, 11, -2, 2, 7]

Two inputs

352

So how do we build functions that take in more than one input at once?

Functions with multiple inputs

```python
def add_lists(a_list, b_list):

    sum_list = []

    for index in range(len(a_list)):

        sum = a_list[index] + b_list[index]

        sum_list.append(sum)

    return sum_list
```

Multiple inputs are separated by commas

The Python syntax for multiple inputs is much the same as it is for a mathemtical function: we separate the inputs by commas.

Functions with multiple inputs

```
def add_lists(a_list, b_list):

    sum_list = []

    for index in range(len(a_list)):

        sum = a_list[index] + b_list[index]

        sum_list.append(sum)

    return sum_list
```
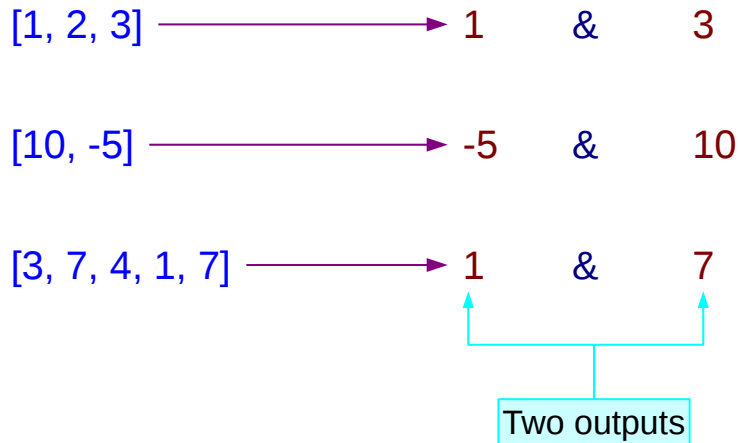
We have two lists…

…so we have to use indexing

354

Note that functions that take in more than one list typically need to use indices.

Multiple outputs

Write a function to find minimum *and* maximum value in a list

[1, 2, 3] ⟶ 1 & 3

[10, -5] ⟶ -5 & 10

[3, 7, 4, 1, 7] ⟶ 1 & 7

Two outputs

But what if we want to return multiple values?

We can write a function that determines the minimum value in a list, and we can write a function that returns the maximum. What do we do if we want to find both?

Finding just the minimum

```python
def min_list(a_list):

    min_so_far = a_list[0]      ← List cannot be empty!

    for a in a_list:

        if a < min_so_far:
            min_so_far = a


    return min_so_far           ← Returning a single value
```

minlist.py

356

So here's the function that determines the minimum value in a list…

Finding just the maximum

```python
def max_list(a_list):

    max_so_far = a_list[0]

    for a in a_list:

        if a > max_so_far:         ← Only real change
            max_so_far = a


    return max_so_far              ← Returning a single value
```

357

…and just the maximum.

Finding both

```
def minmax_list(a_list):

    min_so_far = a_list[0]
    max_so_far = a_list[0]

    for a in a_list:

        if a < min_so_far:
            min_so_far = a

        if a > max_so_far:
            max_so_far = a


    return what?
```

This is the real question

Combining the bodies of these two functions is quite straightforward.
But what do we return?

Returning both

```
def minmax_list(a_list):

    min_so_far = a_list[0]
    max_so_far = a_list[0]

    for a in a_list:

        if a < min_so_far:
            min_so_far = a

        if a > max_so_far:
            max_so_far = a


    return min_so_far, max_so_far
```

A *pair* of values

Two return two values we simply put them both after the `return` statement separated by a comma, just as we would have done with the inputs.

"Tuples"

e.g.   `min_value, max_value`   Pair

`min_value, avg_value, max_value` Triple

Commas

Often written with parentheses:

`(min_value, max_value)`

`(min_value, avg_value, max_value)`

These sets of values separated by commas (but not in square brackets to make a list) are called "tuples" in Python. Sometimes they are written with round brackets around them to make it clearer that they come together. But it's the comma that is the active ingredient making them a tuple, not the brackets.

Using tuples to return values

```
def …                                    In the function definition

    return  min_value , max_value




                                         Using the function

minimum , maximum  = minmax_list(values)
```
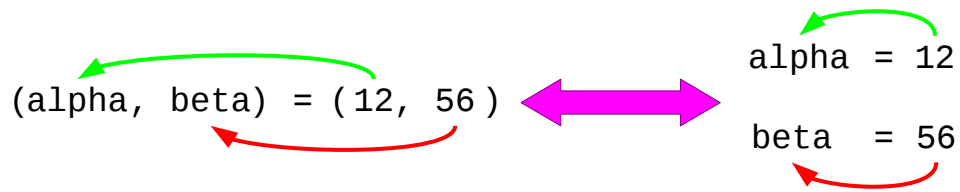
If we return a pair of values in a tuple, we can also attach a pair of names to them as a tuple too.

Using tuples to attach names

```
                                    alpha = 12
(alpha, beta) = (12, 56)  ⟷
                                    beta  = 56
```

We can do this outside the context of functions returning values, of course.
We can do it anywhere.

Swapping values

```
>>> alpha = 12

>>> beta = 56

>>> (alpha, beta) = (beta, alpha)    Swapping values

>>> print(alpha)

56

>>> print(beta)

12
```

Because the entire right hand side is evaluated before the left hand side is considered this lets us use tuples for some particularly useful tricks. perhaps the most useful is swapping two values.

Assignment works right to left

```
alpha = 12

beta = 56

(alpha, beta) = (beta, alpha)
```

Stage 1:   (beta, alpha) ⟶ (56, 12)

Stage 2:   (alpha, beta) = (56, 12)

The values associated with the names are evaluated first. Then the names get reattached to those values, regardless of what names they might have had before.

Progress

Multiple inputs
$$\texttt{def thing(in}_1\texttt{, in}_2\texttt{, in}_3\texttt{):}$$

Multiple outputs
$$\texttt{return (out}_1\texttt{, out}_2\texttt{, out}_3\texttt{)}$$

"Tuples"
```
(a, b, c)
```

Simultaneous assignment
```
(a, b) = (a+b, a-b)
```

365

Exercise 19

The script `exercise19.py` is an answer to exercise 16.
Edit it to:

1. define a function `file_stats()` that takes a file
   name and returns a triple `(n_lines, n_words, n_chars)`

2. use `input()` to read in a file name

3. use `file_stats()` with that file name

4. end with `print(filename, file_stats(filename))`

10 minutes

Tuples and lists: similarities

```
>>> x = ['alpha','beta']      >>> y = ('alpha','beta')

>>> x[1]                       >>> y[1]
'beta'        Indexing          'beta'


>>> len(x)                     >>> len(y)
2             Length            2


>>> (a, b) = (1, 2)            >>> [c, d] = [1, 2]
>>> a                          >>> c
1             Simultaneous      1
              asignment
```

Now that we have seen how tuples work we should take a moment to observe that they are very much like lists. So why don't we just use lists?

There are very many properties that lists and tuples share.

Tuples and lists: differences

```
>>> x = ['alpha','beta']          >>> y = ('alpha','beta')

>>> x[1] = 'B'                     >>> y[1] = 'B'
                                   TypeError:
>>> x                             'tuple' object does not
['alpha','B']                      support item assignment
```

Lists are *mutable*

Tuples are *immutable*

But there is one critical difference. Like strings, tuples are immutable.

Tuples and lists: philosophy

       Tuples

Sequential:
Concept of "next item"

Simultaneous:
All items "at once"

Best with all items
of the same type

Safe to have
multiple types

Serial

Parallel

```
[2, 3, 5, 7, 11, 13,
 17, 19, 23, 29, 31]
```

```
('Dowling', 'Bob',
 50, 105.0, 'rjd4')
```

Sequence of prime numbers

Surname, forename,
age, weight, user id

369

Philosophically, it is appropriate to use lists where there is a sequence of items or where you *need* mutability. Tuples are more appropriate for circumstances where you think about all the items at once.

Because of this, tuples are appropriate where you want to join together a collection of parameters of different types into a single object.

Functions we have written so far

```
total(list)
```

```
add_lists(list₁,list₂)
```

```
minmax_list(list)
```

To date we have written a small number of functions ourselves.

Once we become serious Python programmers using the computer for our day job then we would expect to write many more.

Reusing functions within a script

```
def square(limit):
    ...

...

squares_a = square(34)

...

five_squares = squares(5)

...

squares_b = squares(56)

...
```

One definition

Multiple uses in
the same file

Easy!

Within a script reusing a function is easy. We simply call the function
whenever we want it.

Reusing functions between scripts?

```
def squares(limit):
    ...
```

One definition

```
...
squares_a = squares(34)
...
```

Multiple uses in
multiple files

```
five_squares = squares(5)

...
```
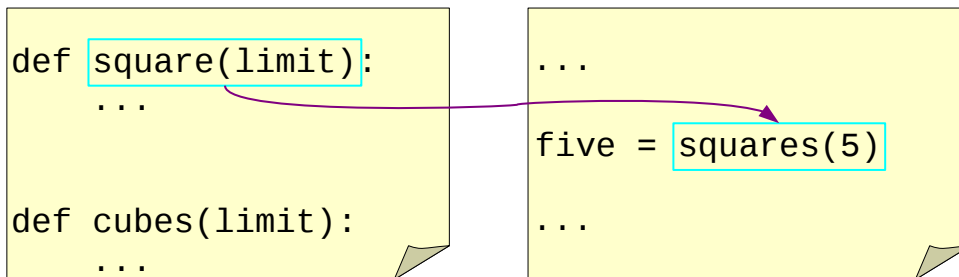
```
...
squares_b = squares(56)
...
```

How?

But what happens if we want to use a function in more than one script?

"Modules"

```
def square(limit):          ...
    ...
                            five = squares(5)

def cubes(limit):           ...
    ...
```
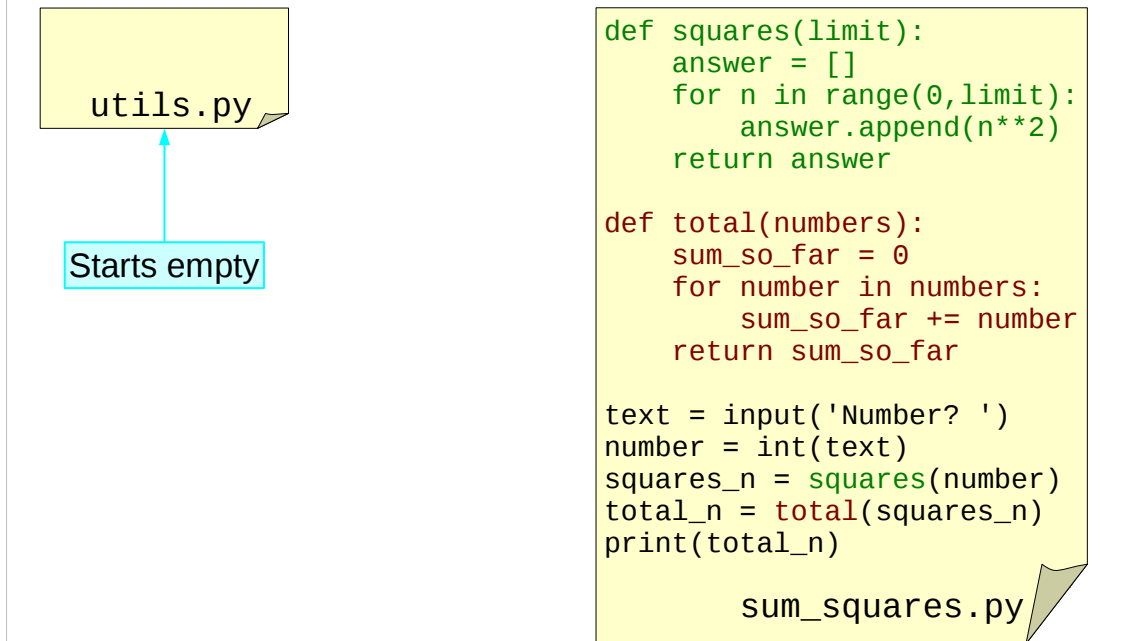
Definition ——————————————▶ Use

Module: a container of functions

373

Python has a mechanism to assist with this called "modules". A module is a collection of functions (and other material) which can then be imported into a script and used within that script. If we can write our own module with our own functions then we can import them into our own scripts.

Modules: a worked example — 1a

utils.py

Starts empty

```python
def squares(limit):
    answer = []
    for n in range(0,limit):
        answer.append(n**2)
    return answer

def total(numbers):
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number
    return sum_so_far

text = input('Number? ')
number = int(text)
squares_n = squares(number)
total_n = total(squares_n)
print(total_n)
```

sum_squares.py

We will start with a file called `sum_squares.py` which uses two functions to add up the squares of numbers from zero to some limit. We want to transfer those function definitions into a different file which we will call `utils.py` (which starts empty) but still be able to use them in our original file.

Modules: a worked example — 1b

```
$ python3 sum_squares.py

Number? 5

30                          = 0 + 1 + 4 + 9 + 16


$ python3 sum_squares.py

Number? 7

91                          = 0 + 1 + 4 + 9 + 16 + 25 + 36
```

Just to prove I'm not fibbing, here it is working before we move anything about.

Modules: a worked example — 2a

```python
def squares(limit):
    answer = []
    for n in range(0,limit):
        answer.append(n**2)
    return answer

def total(numbers):
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number
    return sum_so_far
```

utils.py

```python
text = input('Number? ')
number = int(text)
squares_n = squares(number)
total_n = total(squares_n)
print(total_n)
```

sum_squares.py

Move the definitions
into the other file.

376

Using the text editor we move the definitions from `sum_squares.py` to
`utils.py`.

Modules: a worked example — 2b

```
$ python3 sum_squares.py

Number? 5

Traceback (most recent call last):
  File "sum_squares.py", line 4, in <module>
    squares_n = squares(number)
NameError: name 'squares' is not defined
```

Because we have (re)moved its definition.

Unsurprisingly, this breaks `sum_squares.py`.

Modules: a worked example — 3a

```python
def squares(limit):
    answer = []
    for n in range(0,limit):
        answer.append(n**2)
    return answer

def total(numbers):
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number
    return sum_so_far
```
utils.py

```python
import utils

text = input('Number? ')
number = int(text)
squares_n = squares(number)
total_n = total(squares_n)
print(total_n)
```
sum_squares.py

```
import utils
and *not*
import utils.py
```

import: Make
a reference to
the other file.

378

We need to import the functioned defined in `utils.py` into
`sum_squares.py`.

First, we add the instruction "`import utils`" to the top of the
`stm_squares.py` file.

Note that we import "`utils`", not "`utils.py`".

Modules: a worked example — 3b

```
$ python3 sum_squares.py

Number? 5

Traceback (most recent call last):
  File "sum_squares.py", line 4, in <module>
    squares_n = squares(number)
NameError: name 'squares' is not defined
```

Still can't find the function(s).

On its own this is not sufficient.

Modules: a worked example — 4a

```
squares()  ──────▶

utils.squares()


total()    ──────▶

utils.total()
```

```
import utils


text = input('Number? ')
number = int(text)
squares_n = utils.squares(number)
total_n = utils.total(squares_n)
print(total_n)


              sum_squares.py
```

utils.….: Identify the functions as coming from the module.

We have to indicate to Python that these functions it is looking for in `sum_squares.py` come from the `utils` module. To do this we include "`utils.`" at the start of their names.

Modules: a worked example — 4b

```
$ python3 sum_squares.py

Number? 5

30


$ python3 sum_squares.py

Number? 7

91
```

Working again! ☺

And now it works.

Progress

Sharing functions between scripts

"Modules"

Importing modules   `import` *module*

Using functions from modules   *module*`.function()`

382

Exercise 20

The script exercise20.py is an answer to exercise19.

Move the function `file_stats()` from exercise19.py
into utils.py and edit exercise19.py so that it still works.

5 minutes

383

The Python philosophy

A small core language …

… plus lots of modules

"Batteries included"

math

sys

string

384

We have met the majority of the Python *language* already! But obviously Python has facilities to do much more than we have seen so far. The trick is that Python comes with a large number of modules of its own which have functions for performing no end of useful things.

This philosophy is called "batteries included". You probably already have the module you need to do your specialist application.

Example: the "math" module

```
>>> import math
```
Load in the "math" module.

Run the `sqrt()` function...

```
>>> math.sqrt(2.0)
```
... from the `math` module.

```
1.4142135623730951
```

Let's see an example of an "included battery". At the very start of this course we write ourselves a square root program. Now let's see what Python offers as an alternative.

We import the "`math`" module. (No trailing "`s`"; this is the American spelling.)

In the `math` module is `a sqrt()` function. We can access this as `math.sqrt()`.

Most of the fundamental mathematical operations can be found in the `math` moWe will see how to find out exactly what is in a module in a few slides' time.

```
import module as alias
```

```
>>> import math
```
Too long to keep typing?
```
>>> math.sqrt(2.0)

1.4142135623730951


>>> import math as m
```
"Alias"
```
>>> m.sqrt(2.0)

1.4142135623730951
```

There are those who object to typing. If "`math`" is too long then we can use an aliasing trick to give the module a shorter name.
(The problem is rarely with `math`. There is a built-in module called "`multiprocessing`" though which might get tiresome.)

Don't do these

```
>>> from math import sqrt

>>> sqrt(2.0)

1.4142135623730951
```

Much better to track the module.

!

```
>>> from math import *

>>> sqrt(2.0)

1.4142135623730951
```

!!

python does permit you to do slightly more than that. You can suppress the name of the module altogether.

You are beginners so please take it from an old hand on trust that this turns out to be a very bad idea. You want to keep track of where your functions came from!

What system modules are there?

Python 3.2.3 comes with over 250 modules.

glob	math	argparse	csv
io	cmath	datetime	html
os	random	getpass	json
signal	colorsys	logging	re
subprocess	email	pickle	string
sys	http	sqlite3	unicodedata
tempfile	webbrowser	unittest	xml

388

There are many modules that come with Python.

"Batteries included"

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

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

CDROM          binascii    inspect      shlex
```
263 modules
```
bdb            importlib   shelve

Enter any module name to get more help.  Or,
type "modules spam" to search for modules whose
descriptions contain the word "spam".
```
Not quite this simple

To find out exactly what modules come with your version of Python ask the help system.

A word of warning, though. The text at the bottom "Enter any module name…" is not quite right.

If you give the `help()` command with no argument then you are dropped into an interactive help system. There you can type the name of a module or type "`modules spam`", etc.

```
>>> help()

Welcome to Python 3.1!  This is the online help utility.
…
help> modules subprocess

Here is a list of matching modules.  Enter any module
name to get more help.

subprocess - subprocess - Subprocesses with accessible
I/O streams

help> quit

>>>
```

Additional downloadable modules

Numerical	Databases	
numpy	pyodbc	
scipy	psycopg2	PostgreSQL
	MySQLdb	MySQL
	cx_oracle	Oracle
Graphics	ibm_db	DB2
matplotlib	pymssql	SQL Server

390

But, of course, there's never the particular module *you* want. There are modules provided by people who want Python to interoperate with whatever it is they are offering.

There are three sets of additional modules that you may end up needing to know about.

The numerical and scientific world has a collection of modules called Numerical Python ("numpy") and "scientific python" ("scipy") which contain enormous amounts of useful functions and types for numerical processing.

Every database under the sun offers a module to let Python access it.

Finally there is a module to offer very powerful 2D graphics for data visualisation and presentation.

An example system module: sys

```
import sys

print(sys.argv)
```

argv.py

```
$ python3 argv.py one two three

['argv.py', 'one', 'two', 'three']
```

index | 0 | 1 | 2 | 3

```
$ python3 argv.py 1 2 3

['argv.py', '1', '2', '3']
```

Always strings

We will take a brief look at another commonly used module to illustrate some of the things Python has hidden away in its standard set.

The "sys" module contains many **sys**tems-y things. For example, it contains a list called sys.argv which contains the **arg**ument **v**alues passed on the command line when the script was launched.

Note two things:

1.	that item zero in this list is always the name of the script itself,
2.	the items are always strings

sys.exit()

`exit()` ← What we have been using

`sys.exit(`*`rc`*`)` ← What we should use

↑ "Return Code": an integer

0: Everything was OK

≠0: Something went wrong

Also tucked away in the `sys` module is the `sys.exit()` function.

Up to this point we have been using the `exit()` function to quit our scripts early. However this function is something of a filthy hack and `sys.exit()` provides superior quitting and an extra facility we will be able to make use of.

The `sys.exit()` function takes an integer argument. This is the program's "return code" which is a very short message back to the operating system to indicate whether the program completed successfully or not. A return code of 0 means "success". A non-zero return code means failure. Some programs use different non-zero codes for different failures but many (most?) simply use a value of 1 to mean "something went wrong".

If your script simply stops because it reached the end then Python does an automatic `sys.exit(0)`.

An example system module: sys

`sys.modules`   All modules currently imported

`sys.path`   Directories Python searches for modules

`sys.version`   Version of Python

`sys.stdin`   Where input inputs from

`sys.stdout`   Where print prints to

`sys.stderr`   Where errors print to

`sys.float_info`   All the floating point limits

393

And there's plenty more…

Modules in Python

"How do I do
 X in Python?"

"What's the Python
 module to do X?"

"Where do I find
 Python modules?"

So the Python philosophy places a lot of functionality into its modules.

This means that we have to be able to find modules and know what they can do.

Finding modules

Python: Built-in modules

SciPy: **Sci**entific **Py**thon modules

PyPI: **Py**thon **P**ackage **I**ndex

Search: "Python3 module for X"

Some useful URLs:

`http://docs.python.org/py3k/py-modindex.html`

This contains the list of all the "batteries included" modules that come with Python. For each module it links through to their documentation.

`http://www.scipy.org/Topical_Software`
`http://numpy.scipy.org/`

Scientific Python contains very many subject-specific modules for Python. Most depend on the Numerical Python module `numpy`.

`http://pypi.python.org/pypi`   (do check for Python3 packages)

This is the semi-official dumping ground for everything else.

`http://www.google.co.uk/`

And for everything else there's Google (who are big Python users, by the way).

Help with modules

```
>>> import math

>>> help(math)

NAME
    math

DESCRIPTION
    This module is always available.  It provides
    access to the mathematical functions defined
    by the C standard.

...
```

I promised information on how to find out what is in a module. Here it is.
Once a module has been imported you can ask it for help.

Help with module functions

```
…

FUNCTIONS

  acos(x)
    Return the arc cosine (measured in
    radians) of x.
…


>>> math.acos(1.0)

0.0
```

The help will always include information on every function in the module…

Help with module constants

```
...

DATA
    e = 2.718281828459045
    pi = 3.141592653589793
...




>>> math.pi

3.141592653589793
```

…and every data item.

Help for our own modules?

```
def squares(limit):
    answer = []
    for n in range(0,limit):
        answer.append(n**2)
    return answer

def total(numbers):
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number
    return sum_so_far
```

utils.py

Basic help already provided by Python

```
>>> import utils

>>> help(utils)

NAME
    utils

FUNCTIONS
    squares(limit)

    total(numbers)

FILE
    /home/y550/utils.py
```

399

What help can we get from our own module?

By now you should have a `utils.py` file with some functions of your own in it. The help simply lists the functions the module contains and the file it is defined in.

Adding extra help text

```
"""Some utility functions
from the Python for
Absolute Beginners course
"""

def squares(limit):
    answer = []
    for n in range(0,limit):
        answer.append(n**2)
    return answer

def total(numbers):
    sum_so_far = 0
    for number in numbers:
        sum_so_far += number
    return sum_so_far
```

utils.py

```
>>> import utils    Fresh start

>>> help(utils)

NAME
    utils

DESCRIPTION
    Some utility functions
    from the Python for
    Absolute Beginners course

FUNCTIONS
    squares(limit)

    total(numbers)
```

400

But we can do better than that.

If we simply put a Python string (typically in long text triple quotes) at the top of the file before any used Python (but after comments is fine) then this becomes the description text in the help.

Note: You need to restart Python and re-import the module to see changes.

Adding extra help text to functions

```
"""Some utility functions
from the Python for
Absolute Beginners course
"""

def squares(limit):
    """Returns a list of
    squares from zero to
    limit**2.
    """
    answer = []
    for n in range(0,limit):
        answer.append(n**2)
    return answer
```

utils.py

```
>>> import utils      Fresh start

>>> help(utils)

NAME
    utils

DESCRIPTION
    ...

FUNCTIONS
    squares(limit)
        Returns a list of
        squares from zero to
        limit**2.
```

If we put text immediately after a `def` line and before the body of the function it becomes the help text for that function, both in the module-as-a-whole help text…

Adding extra help text to functions

```
"""Some utility functions
from the Python for
Absolute Beginners course
"""

def squares(limit):
    """Returns a list of
    squares from zero to
    limit**2.
    """
    answer = []
    for n in range(0,limit):
        answer.append(n**2)
    return answer
```
utils.py

```
>>> import utils      Fresh start

>>> help(utils.squares)

squares(limit)
    Returns a list of
    squares from zero to
    limit**2.
```

402

…and in the function-specific help text.

Progress

Python a small language…   Functionality ⟶ Module

…with many, many modules

System modules

Foreign modules

Modules provide help   `help(`*`module`*`)`
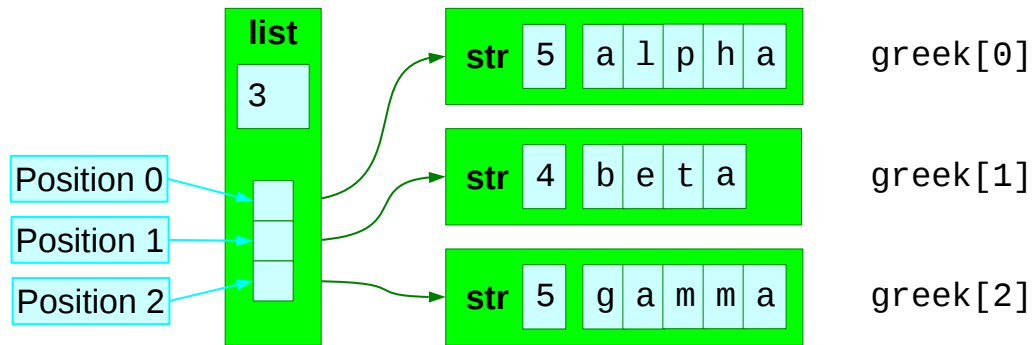
Doc strings   `help(`*`module.function`*`)`

403

Exercise 21

Add help text to your `utils.py` file.

5 minutes

Recap: Lists & Indices

```
greek = ['alpha', 'beta', 'gamma']
```

|	
list	
3	
Position 0 →	**str** 5 a l p h a greek[0]
Position 1 →	**str** 4 b e t a greek[1]
Position 2 →	**str** 5 g a m m a greek[2]

405

We have one last Python type to learn about. To give it some context, we will recap the list type that we have spent so much time using.

A list is basically an ordered sequence of values. The position in that sequence is known as the index.

"Index in — Value out"

Must be
a number

index → list → value

start at 0

no gaps in
sequence

1   greek[1] → 'beta'
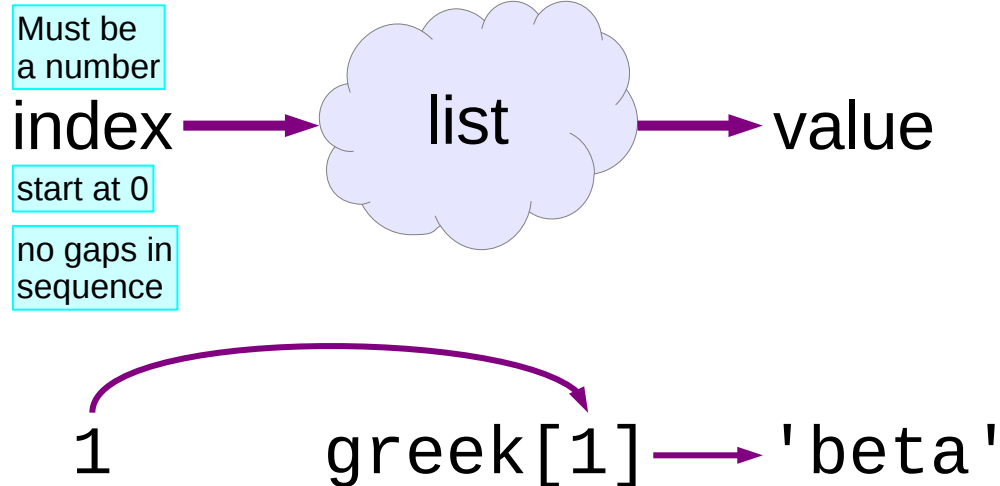
406

If we now forget about the internals of a list, though, we can think of it as "some sort of Python object" that takes in a number (the index) and spits out a value.

Other "indices": Strings?

English → dictionary → Spanish

'cat' ——→ 'gato'

'dog' —— dictionary —→ 'perro'

'mouse' ——→ 'ratón'

COLLINS GEM
SPANISH
DICTIONARY
The best-selling Spanish
mini dictionary
4th edition
SPANISH · ENGLISH
ENGLISH · SPANISH

Can we generalise on this idea by moving away from the input (the index) needing to be a number?

Can we model a dictionary where we take in a string (a word in English, say) and give out a different string (the corresponding word in Spanish, say).

(Note: the author is fully aware that translation is not as simple as this. This is just a toy example.)

Other "indices": Tuples?

(**x**, **y**) ⟶ dictionary ⟶ Objects

(2,5) ⟶ dictionary ⟶ 'Treasure'

(4,5) ⟶ 'Fortress'
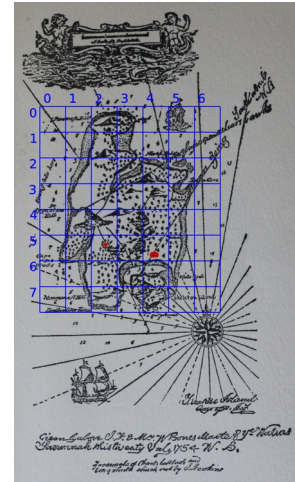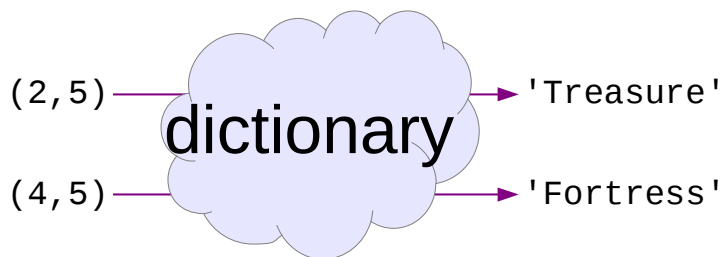
Or, perhaps, pairs of numbers (x,y) in and items on a map out?

Python "dictionaries"

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }


>>> en_to_es['cat']

'gato'
```

Python does have exactly such a general purpose mapper which it calls a "`dict`", short for "**dict**ionary".

Here is the Python for establishing a (very small) English to Spanish dictionary that knows about two words.

We also see the Python for looking up a word in the dictionary.

We will review this syntax in some detail…

Creating a dictionary — 1

Curly brackets

Comma

```
{ 'cat':'gato' , 'dog':'perro' }
```

'cat' → 'gato'   'dog' → 'perro'
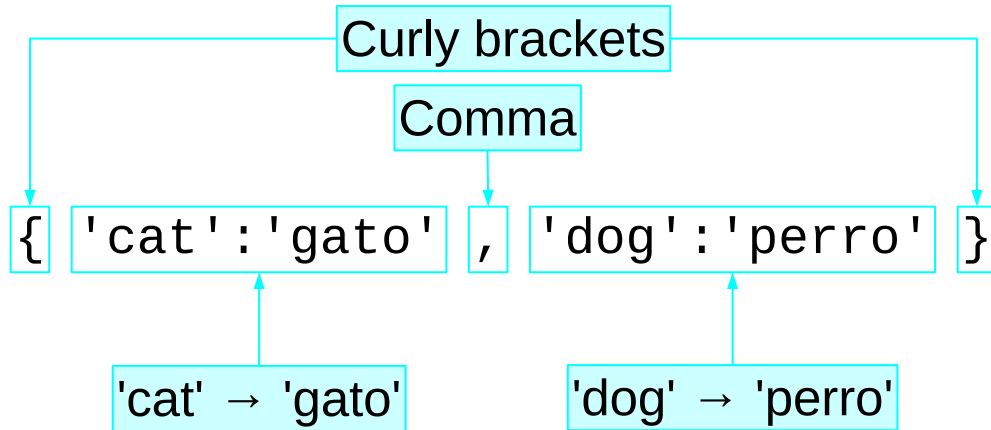
First we will look at creating a dictionary. In the same way that we can create a list with square brackets, we can create a dictionary with curly ones.

Each item in a dictionary is a pair of values separated by a colo

They are separated by commas.n.

Creating a dictionary — 2

'cat' : 'gato'

{ 'cat':'gato' , 'dog':'perro' }

"key" : "value"
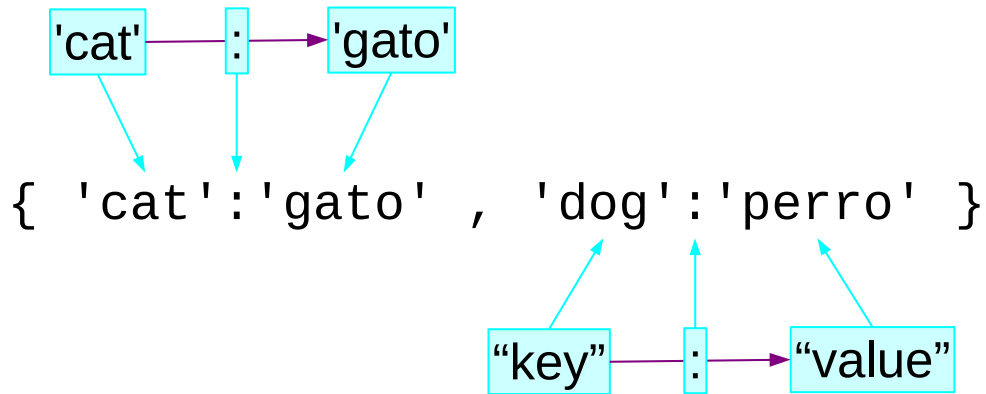
The pairs of items separated by colons are known as the "key" and "value". The key is what you put in (the English word in this example) that you look up in the dictionary and the value is what you get out (the translation into Spanish in this example).

Using a dictionary — 1

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }
```

Creating the dictionary

```
>>> en_to_es['cat']
```
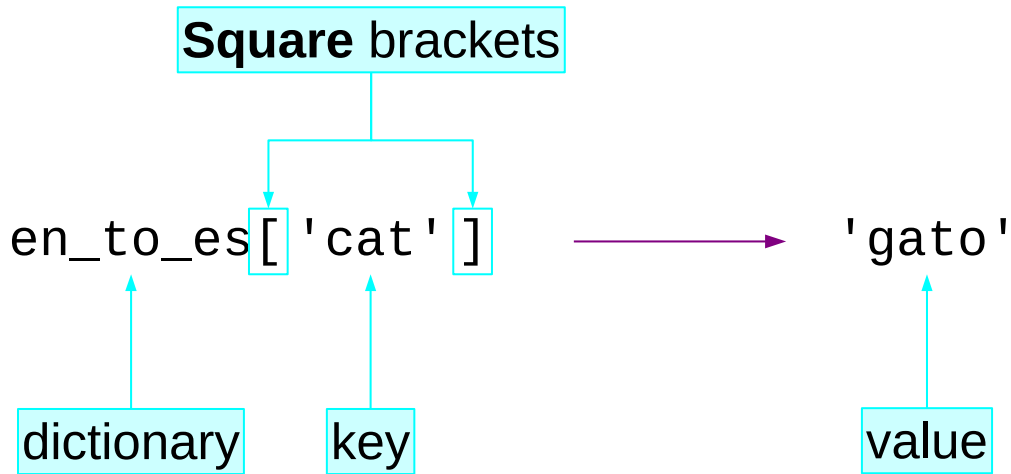
Using the dictionary

```
'gato'
```

412

Now we have seen how to create a (small) dictionary we should look at how to use it.

Using a dictionary — 2

`Square` **brackets**

`en_to_es[ 'cat' ]` ⟶ `'gato'`

dictionary   key   value

413

To look something up in a dictionary we pass it to the dictionary in exactly the same way as we passed the index to a list: in *square* brackets. Curly brackets are just for creating a dictionary; after that it' square brackets again.

Missing keys

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }


>>> en_to_es['dog']

'perro'


>>> en_to_es['mouse']

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

✓

✗

Error message

414

The equivalent to shooting off the end of a list is asking for a key that's not in
a dictionary.

Dictionaries are one-way

> ⚠️ !

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }

>>> en_to_es['dog']          ✓

'perro'

>>> en_to_es['perro']        ✗

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

Looking for a **key**

Also note that dictionaries are *one-way*.

415

Adding to a dictionary

```
>>> en_to_es = { 'cat':'gato' , 'dog':'perro' }
```

Initial dictionary has no 'mouse'

```
>>> en_to_es['mouse'] = 'ratón'
```

Adding 'mouse' to the dictionary

```
>>> en_to_es['mouse']

'ratón'
```
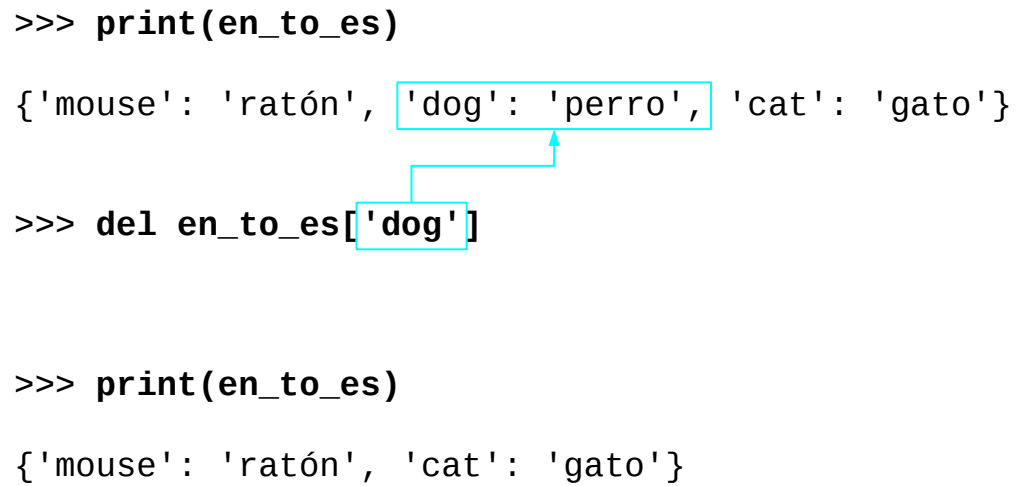
416

Adding key-value pairs to a dictionary is a lot easier than it is with lists. With lists we needed to append on the end of a list. With dictionaries, because there is no inherent order, we can simply define them with a simple expression on the left hand side.

Removing from a dictionary

```
>>> print(en_to_es)

{'mouse': 'ratón', 'dog': 'perro', 'cat': 'gato'}

>>> del en_to_es['dog']


>>> print(en_to_es)

{'mouse': 'ratón', 'cat': 'gato'}
```

We can use `del` to remove from a dictionary just as we did with lists.

Progress

Dictionaries   Key ——→ Value

{ key$_1$:value$_1$, key$_2$:value$_2$, key$_3$:value$_3$ }

Looking up values   `dictionary[key]` ——→ `value`

Setting values   `dictionary[key] = value`

Removing keys   `del dictionary[key]`

418

Exercise 22

Complete exercise22.py to create
an English to French dictionary.

cat ⟶ chat

dog ⟶ chien

mouse ⟶ souris

snake ⟶ serpent

5 minutes

419

What's in a dictionary? — 1

```
>>> en_to_es
{'mouse': 'ratón', 'dog': 'perro', 'cat': 'gato'}


>>> en_to_es.keys()
dict_keys(['mouse', 'dog', 'cat'])


>>> en_to_es.values()
dict_values(['ratón', 'perro', 'gato'])
```

Orders match

Just treat them like lists   (or convert them to lists)

To date we have created our own dictionaries. If we are handed one how do we find out what keys and values are in it?

Dictionaries support two methods which return the sort-of-lists of the keys and values. We mention them here only for completeness.

Don't forget that you can always convert a sort-of-list into a list with the `list()` function.

What's in a dictionary? — 2

```
>>> en_to_es.items()
```
Most useful method

```
dict_items([('mouse', 'ratón'), ('dog', 'perro'),
('cat', 'gato')])
```
(Key,Value) pairs/tuples

```
>>> for (english, spanish) in en_to_es.items():
...     print(spanish, english)
...
ratón mouse
perro dog
gato cat
```

421

By far the best way to get at the contents of a dictionary is to use the `items()` method which generates a sort-of-list of the key-value pairs as tuples. Running a `for…` loop over this list is the easiest way to process the contents of a directory.

What's in a dictionary? — 3

Common simplification

```
>>> list(en_to_es.items())
[('mouse','ratón'), ('dog','perro'),('cat','gato')]
```

Don't be afraid to convert it explicitly into a list. Unless you dictionary is huge you won't see any problem with this.

Getting the list of keys

dictionary $\xrightarrow{\quad \texttt{list()} \quad}$ list of keys

```
{'the': 2, 'cat': 1, 'sat': 1, 'on': 1, 'mat': 1}
```

$\downarrow$

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

Unfortunately when you convert a dictionary directly into a list you get the list of keys not the list of)(key,value) pairs. This is a shame but is a compromise for back compatibility with previous versions.

Is a key in a dictionary?

```
>>> en_to_es['snake']

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'snake'
```
Want to avoid this

```
>>> 'snake' in en_to_es

False
```
We can test for it

Because of this conversion to the list of keys we can ask if a key is in a dictionary using the `in` keyword without having to know its corresponding value.

Example: Counting words — 1

```
words = ['the','cat','sat','on','the','mat']
```



```
counts = {'the':2,'cat':1,'sat':1,'on':1,'mat':1}
```

Let's have a serious worked example.

We might be given a list of words and want to count the words by how many times they appear in the list.

Example: Counting words — 2

```
words = ['the','cat','sat','on','the','mat']

counts = {}

for word in words:
```

counts = {} ← Start with an empty dictionary

for word in words: ← Work through all the words

Do something

We start by creating an empty dictionary. It's empty because we haven't read any words yet.

Then we loop through the list of words using a standard `for…` loop.

For each word we have to do something to increment the count in the dictionary.

Example: Counting words — 3

```
words = ['the','cat','sat','on','the','mat']

counts = {}

for word in words:
    counts[word] += 1
```

This will not work

counter1.py

Unfortunately a simply increment of the value in the dictionary isn't enough.

Why doesn't it work?

```
counts = {'the':1, 'cat':1}
```

✓ `counts['the'] += 1`

The key must already be in the dictionary.

`counts['the'] = ` `counts['the']` ` + 1`

✗ `counts['sat'] += 1`

Key is not in the dictionary!

`counts['sat'] = ` `counts['sat']` ` + 1`

428

We cannot increment a value that isn't there. Until the program meets a word for the first time it has no entry in the dictionary, and certainly not an entry with numerical value 0.

Example: Counting words — 4

```
words = ['the','cat','sat','on','the','mat']

counts = {}

for word in words:

    if word in counts:

        counts[word] += 1

    else:
```

Do something   Need to add the key

So we have to test to see if the word is already in the dictionary to increment it if it is there and to do something else if it is not. Note how we use the "`if key in dictionary`" test.

Example: Counting words — 5

```
words = ['the','cat','sat','on','the','mat']

counts = {}

for word in words:

    if word in counts:

        counts[word] += 1

    else:

        counts[word] = 1

print(counts)
```

counter2.py

That something else is to create it with its initial value of 1 (because we have met the word once now).

Example: Counting words — 6

```
$ python3 counter2.py


{'on': 1, 'the': 2, 'sat': 1, 'mat': 1, 'cat': 1}
```

⚠️ You cannot predict the order of the
keys when a dictionary prints out.

Dictionaries are unordered entities. You cannot predict the order that the
keys will appear when you print a dictionary or step through its keys.

Example: Counting words — 7

```
print(counts)
```
Too ugly

```
items = list(dictionary.items())
```
Better

```
items.sort()

for (key, value) in items:

    print(key, value)
```

counter3.py

Simply printing a dictionary gives ugly output.

We can pull out the (key,value) pairs and print them individually if we want.

Notice the use of pulling out the items, converting them into a list and then sorting them.

Example: Counting words — 8

```
$ python3 counter3.py


cat 1
mat 1
on 1
sat 1
the 2
```

Progress

Inspection methods
```
dictionary.keys()
dictionary.values()
dictionary.items()
```
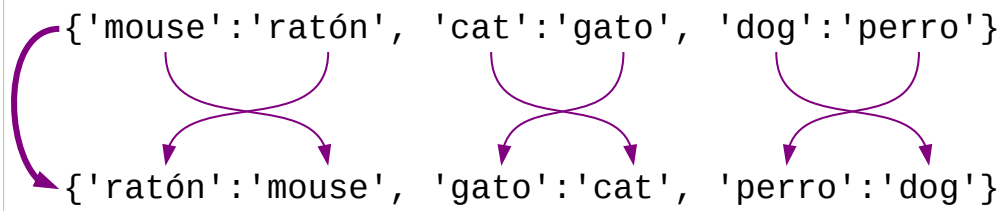
Testing keys in dictionaries
```
if key in dictionary:
        ...
```

Creating a list of keys
```
keys = list(dictionary)
```

434

Exercise 23

Complete exercise23.py to write
a script that reverses a dictionary.

```
{'mouse':'ratón', 'cat':'gato', 'dog':'perro'}



{'ratón':'mouse', 'gato':'cat', 'perro':'dog'}
```

5 minutes

Formatted output

```
$ python3 counter3.py

cat 1
mat 1
on 1
sat 1
the 2
```

on 1 ← Ugly!

```
cat 1
mat 1
on  1
sat 1
the 2
```

We want data
nicely aligned

436

We have one last topic to cover. The output from our word counter is not as pretty as it might be. The last topic we will cover is Python text formatting.

Full details of the formatting system can be found online at
`docs.python.org/py3k/library/string.html#formatspec`

The format() method

```
>>> 'xxx{}yyy{}zzz'.format('A', 100)

'xxxAyyy100zzz'
```

'xxx{}yyy{}zzz'

'xxxAyyy100zzz'

In Python 3 the string type has a method called `format()` which takes arguments and returns another string which is the original with the method's arguments inserted into it in certain places. Those places are marked with curly brackets in the string.

Curly brackets are otherwise quite normal characters. It's only the `format()` method that cares about them.

In its simplest form, the `format()` method replaces each pair of curly brackets with the corresponding argument.

String formatting — 1

```
>>> 'xxx{:5s}yyy'.format('A')
```

'xxxA⎵⎵⎵⎵yyy'

'xxx{:5s}yyy'   {:5s}

'xxxA⎵⎵⎵⎵yyy'   s — substitute a **string**

5 — pad to **5** spaces
(left aligns by default) 438

The real fun starts when we put something inside those curly brackets. These are the formatting instructions.

The simplest examples start with a colon followed by some layout instructions. (We will see what comes in front of the colon in a few slides time.) The number indicates how many spaces to allocate for the insertion and the letter that follows tells it what type ob object to expect. "s" stands for **s**tring.

String formatting — 2

```
>>> 'xxx{:<5s}yyy'.format('A')
```

'xxxA⎵⎵⎵⎵yyy'

'xxx{:<5s}yyy'   {:<5s}

'xxxA⎵⎵⎵⎵yyy'   < — align to the left (←)

By default strings align to the left of their space. We can be explicit about this by inserting a left angle bracket, which you can think of as an arrow head pointing the direction of the alignment.

String formatting — 3

```
>>> 'xxx{:>5s}yyy'.format('A')
```

'xxx␣␣␣␣Ayyy'

'xxx{:>5s}yyy'   {:>5s}

'xxx␣␣␣␣Ayyy'   > — align to the right (→)

440

If we want right aligned strings then we have to use the alignment marker.

Integer formatting — 1

```
>>> 'xxx{:5d}yyy'.format(123)
```

'xxx␣␣123yyy'

'xxx{:5d}yyy'   {:5d}

'xxx␣␣123yyy'

d — substitute an integer
                    (**d**igits)
5 — pad to **5** spaces
        (right aligns by default)

441

If we change the letter to a "d" we are telling the `format()` method to insert an integer ("**d**igits"). These align to the right by default.

Integer formatting — 2

```
>>> 'xxx{:>5d}yyy'.format(123)
```

`'xxx␣␣123yyy'`

`'xxx{:>5d}yyy'`   `{:>5d}`

`'xxx␣␣123yyy '`

> — align to the right (→)

442

We can be explicit about this if we want.

Integer formatting — 3

```
>>> 'xxx{:>5d}yyy'.format(123)
```

'xxx␣␣123yyy'

'xxx{:<5d}yyy'   {:<5d}

'xxx123␣␣yyy'

< — align to the left (←)

443

And we have to be explicit to override the default.

Integer formatting — 4

```
>>> 'xxx{:05d}yyy'.format(123)

'xxx00123yyy'
```

'xxx{:05d}yyy'   {:05d}

'xxx00123yyy'

0 — pad with zeroes

If we precede the width number with a zero then the number is padded with zeroes rather than spaces and alignment is automatic.

Integer formatting — 5

```
>>> 'xxx{:+5d}yyy'.format(123)
```

'xxx␣+123yyy'

'xxx{:+5d}yyy'   {:05d}

'xxx␣+123yyy '   + — always show sign

If we put a plus sign in front of the number then its sign is always shown, even if it is positive.

Integer formatting — 6

```
>>> 'xxx{:+05d}yyy'.format(123)
```

'xxx␣+0123yyy'

'xxx{:+05d}yyy'   {:05d}

'xxx+0123yyy'

\+ — always show sign

0 — pad with zeroes

And we can combine these.

Integer formatting — 7

```
>>> 'xxx{:5,d}yyy'.format(1234)

'xxx1,234yyy'
```

'xxx{:5,d}yyy'   {:5,d}

'xxx1,234yyy'   , — 1,000s

447

Adding a comma between the width and the "d" adds comma breaks to large numbers.

Floating point formatting — 1

```
>>> 'xxx{:5.2f}yyy'.format(1.2)

'xxx 1.20yyy'
```

'xxx{:5.2f}yyy'   {:5.2f}

'xxx␣1.20yyy'

f — substitute a **f**loat

5 — **5** places *in total*

.2 — **2** places after the point

Floating point numbers are slightly more complicated. The width parameter has two parts, separated by a decimal point. The first number is the width of the entire number (just as it is for strings and integers). The number after the decimal point is the number of decimal points of precision that should be included in the formatted output.

Floating point formatting — 2

```
>>> 'xxx{:f}yyy'.format(1.2)

'xxx1.200000yyy'
```

'xxx{:f}yyy'   {:f}

'xxx1.200000yyy'   {:.6f}

The default is to have six decimal points of precision and to make the field as wide as it needs to be.

Ordering and repeats — 1

```
        0    1    2
>>> 'X{:s}X{:d}X{:f}X'.format('abc', 123, 1.23)
                            0      1     2
'XabcX123X1.230000X'
   0   1      2
```

↓ Equivalent

```
>>> 'X{0:s}X{1:d}X{2:f}X'.format('abc', 123, 1.23)
                              0      1     2
'XabcX123X1.230000X'
   0   1      2
```

450

What comes before the colon?

This is a selection parameter detailing what argument to insert.

The arguments to `format()` are given numbers starting at zero. We can put these numbers in front of the colon.

Ordering and repeats — 2

```
>>> 'X{0:s}X{2:f}X{1:d}X'.format('abc', 123, 1.23)
                                  0      1     2
'XabcX1.230000X123X'
   0        2        1
```

```
>>> 'X{0:s}X{1:d}X{1:d}X'.format('abc', 123, 1.23)
                                  0      1     2
'XabcX123X123X'
   0    1    1
```

We can also use them to change the order that items appear in or to have them appear more than once (or not at all).

Formatting in practice

```
...
formatting = '{:3} {:1}'

for (word, number) in items:
    print(formatting.format(word,number))


$ python3 counter4.py

cat 1
mat 1
on  1
sat 1
the 2
```

The script `counter4.py` is the same as counter3.py but with the new formatting for its output.

Progress

Formatting	`string.format(args)`
Numbered parameters	`{0}` `{1}` `{2}`
Substitutions	`{:>6s}`
	`{:+06d}`
	`{:+012.7f}`

453

Exercise 24

Complete exercise24.py
to format the output as shown:

```
[(Joe,9), (Samantha,45), (Methuselah,969)]
```

```
Joe           9
Samantha     45
Methuselah 969
```

5 minutes

454

And that's it! (And "it" is a lot!)

Text	"if" test	Reading files
Prompting	Indented blocks	Writing files
Numbers	Lists	Functions
Arithmetic	Indices	"Structured programming"
Comparisons	Object methods	Tuples
Booleans	Built-in help	"for" loops
Variables	"for" loops	Modules
Deleting names	"Treat it like a list…"	Dictionaries
"while" loop	Values direct/via index	Formatting

455

And congratulations!

You have completed an introductory course on Python. Well done.

It is only an introductory course and there is more. But do not let that dishearten you; just take a look at what you have accomplished. You now have a firm grounding to go further with Python or to start learning other programming languages. (But the author would like you to stick with Python.)

But wait! There's more…
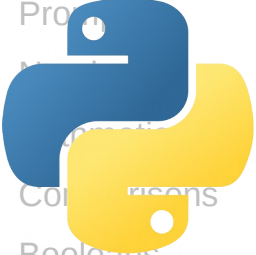
Advanced topics: Self-paced introductions to modules

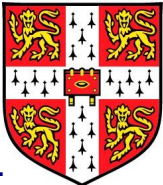Object-oriented programming in Python

If you do want more Python the UCS offers a selection of self-paced courses on some additional language features and on various Python modules to let you learn how to use Python for a specific purpose. We also offer a taught course introducing you to the world of object-oriented programming where you get to write your own methods and types.

Congratulations!

Text	"if" test	Reading files
Prompt	Indented blocks	Writing files
Numbers	Lists	Functions
Formatting	Indices	"Structured programming"
Comparisons	Object methods	Tuples
Booleans	Built-in help	"for" loops
Variables	"for" loops	Modules
Deleting names	"Treat it like a list…"	Dictionaries
"while" loop	Values direct/via index	Formatting

Please don't forget the feedback.

So thank you and congratulations again.