PHAS1240 Session 1 Script

September 16, 2015
Louise Dash (louise.dash@ucl.ac.uk)

1. PHAS1240 Session 1: Basics

1.1 Learning objectives:

By then end of this session, you should:

- Understand the basics of computer arithmetic, operators and data types
- Understand the concept of a variable
- Be able to read and understand a simple linear program
- Be able to correct common errors in a simple program
- Start to understand the principles of good programming

1.2 How to use this document

Follow along by entering commands shown into Spyder using either the console pane or editor window as appropriate. Experiment as well!

2. Fundamental Concepts

2.1 Data types

In computational terms, different types of numbers or data structures are stored and treated differently, so we need to know a bit about the different data types Python (and most other programming languages) uses.

The most common types we will be using are:

- Integers, e.g. 2, 42, 513
- Floating point numbers or floats, e.g 2.5, 6.0, or in exponential notation 1.602E10-16
- Strings: a "string" of alphanumeric characters, often used for text, which are usually surrounded by single or double quotation marks, e.g. "Hello", 'Goodbye'

- Boolean, which can have only two values: True or False (note the initial capitalisation)

Python has an inbuilt command to tell us what type a certain data object is:

```
In [1]: type(2.3)
```

```
Out[1]: float
```

```
In [2]: type(3)
```

```
Out[2]: int
```

```
In [3]: type("python")
```

```
Out[3]: str
```

2.2 Arithmetic operators

We also need to know about the basic arithmetic operations in Python. Most of these are intuitive, and do exactly what you'd think:

- + addition, concatenation
- - subtraction
- * multiplication
- ** exponentiation (n.b. Python, like most computer languages, uses ** for exponentiation, unlike Excel, which uses a ^ symbol)

```
In [4]: 2 + 3.5
```

```
Out[4]: 5.5
```

```
In [5]: 10 - 5
```

```
Out[5]: 5
```

```
In [6]: 2.0 * 4.2
```

```
Out[6]: 8.4
```

```
In [7]: 2.0**8
```

```
Out[7]: 256.0
```

Some we need to be a bit careful about though, such as

- / division

```
In [8]: 4/2
```

```
Out[8]: 2
```

. . . exactly what we expect

```
In [9]: 10.0 / 3.0
```

```
Out[9]: 3.3333333333333335
```

Here we see a tiny error in the final decimal place - this is one of the effects of using floating point numbers, which we will discuss more later.

```
In [10]: 5 / 2
```

```
Out[10]: 2
```

... **Huh?!??**

This is the easiest, and most surprising way that you can trip yourself up in Python if you're not careful.

The division operator / will divide two numbers, but when it is used to divide two *integers*, it will give an *integer* result. If you want to be certain of getting a floating-point output (which you usually will!), make sure that one of the inputs is also a float. For example:

```
In [11]: 5.0 / 2
```

```
Out[11]: 2.5
```

```
In [12]: 5 / 2.0
```

```
Out[12]: 2.5
```

Another thing you need to be aware of, is that Python will sometimes interpret operators differently depending on the context. + acts as an addition operator when used on numbers, but also as a concatenation operator when used on text strings. This is known as an "overloaded" operator as it can perform different functions depending on the context.

For example, let's add together two strings, "2", and "3"

```
In [13]: "2" + "3"
```

```
Out[13]: '23'
```

This is quite useful sometimes though:

```
In [14]: "You" + "Tube"
```

```
Out[14]: 'YouTube'
```

2.3 Operator Precedence

As in normal maths, we have to be careful in which order the operators are applied - in general the rules are the same. Sometimes it's a good idea to add some space in long expressions to make them clearer for humans to read though, for example, which of these do you think is easier to read?

```
In [15]: 5-(4+2)**2/(2.3**3)+5.3*2
```

```
Out[15]: 12.641176954056053
```

```
In [16]: 5 - (4+2)**2 / (2.3**3 )+ 5.3*2
```

```
Out[16]: 12.641176954056053
```

2.4 Variables

If we want to do more useful things with Python, we need to be able to use it as more than just a calculator. In order to do this, we need to introduce another fundamental concept: variables. These are essentially labels or names for an object, and they are assigned using the *assignment operator*, =.

For example, this assigns the value 3 to the variable named x:

```
In [17]: x = 3
```

Now if we ask Python what the value of x is, it will tell us:

```
In [18]: x

Out[18]: 3
```

This is subtly different from the way the statement $x = 3$ would work in algebra though:

The = symbol acts as an operator: it binds the name x to the value 3. (In older languages this would be expressed as "let x = 3", which makes it a bit more intuitive)

Because of this, the statement 3 = x is meaningless - Python tries to assign the value x to the variable name 3 . You should try this out in the console to see the error produced.

Also, if you have experience programming in other languages, you should note that Python is unusual in that the variable name does not store the value directly, but acts as a pointer to the memory location holding the value. This can trip you up if you're unlucky.

2.4.1 Time for a break...

Now you should:

- Do the first two Moodle quizzes (the how-to-do-a-quiz quiz, and the quiz on data types)
- If you've been working for around an hour or more, take a 5-10 minute break!
- Watch the next screencast, on how to build our first program.

---

3. Building our first program

We're now at the point where we have most of the building blocks we need to construct our first program - a computer program is just a sequence of commands.

In the screencast, we saw how to build up a code to calculate the length of a vector, from scratch. Here is the point at which we started the code:

```
In [ ]: x = 3
        y = 2.4
        z = 3.5
```

```
length_squared = x**2 + y**2 + z**2

length = sqrt(length_squared)
```

This didn't work, as:

- sqrt() isn't in base python, we need to explicitly import it, either from the math module, or from numpy;
- Within a program, we need include a "print" statement so that Python outputs the answer to the screen.

We corrected these problems and made the code more usable by:

- Adding **comments**: lines of code that start with # are ignored by Python, but are useful for letting a human know what the code is doing (or supposed to be doing!)
- Making the code interactive by asking for input from the user
- Adding a text string to the output line

This is the final code:

```
In [ ]: ################################################
        # Code to calculate the length of a vector     #
        # Louise Dash 03/07/14                         #
        ################################################

        from numpy import sqrt

        # set up components of the vector
        #x = 3
        #y = 2.4
        #z = 3.5

        x = float(raw_input('input the x-component: '))
        y = float(raw_input('input the y-component: '))
        z = float(raw_input("input the z-component: "))

        # calculate the length
        length_squared = x**2 + y**2 + z**2

        length = sqrt(length_squared)

        # output the result to the screen
        print "The length of the vector is", length
```

Things to note:

- raw_input() accepts whatever the user inputs to the terminal, we then need to nest a float() around this to convert the input to a floating point number.
- We've chosen intuitive variable names that are not too long.
- We've included a comment block right at the top of the code that describes what the code does, who wrote it, and when.

- We've put comments in to describe what each section of code does. This makes it easier to read, and easier to find any errors.

3.1 Python modules

We saw in the example above that Python doesn't have many builtin functions for maths. In fact, base Python is (deliberately) quite small in terms of what it can do. If you want to do something more specialised, you will need to import functions from modules, which are just libraries of different functions.

For this course we'll frequently need to use NumPy, short for Numerical Python, which contains lots of useful mathematical functions (more on these next week). Later on we'll also add different modules for things like plotting graphs and creating animations.

3.2 Error messages and the art of debugging

As you've probably noticed, almost as soon as you start writing code, you'll find yourself encountering error messages. This is an inescapable and fundamental part of coding. It's true that a good programmer will try to avoid making errors in the first place (by following good programming practice), but a *great* programmer is one who has also mastered the art of figuring out how to interpret the error messages and fix the underlying problems, or bugs.

(why "bugs"? See http://en.wikipedia.org/wiki/Software_bug#Etymology)

Some bugs are easy to fix, others are subtle and can take longer to spot.

Let's look at some of the most common error messages you'll see in Python:

3.2.1 NameError:

```
In [20]: sin(2.4)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-20-45c4358037d1> in <module>()
----> 1 sin(2.4)


NameError: name 'sin' is not defined
```

This is where Python doesn't recognise the name of a function you've typed. Mostly this will be because either you've mistyped the function name, or because you haven't imported that function from a module. In this case it's easy to fix, by importing the sin function from the NumPy module:

```
In [21]: from numpy import sin
         sin(2.4)
```

```
Out[21]: 0.67546318055115095
```

3.2.2 TypeError:

```
In [22]: sin("five")
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-22-d03e050b0c73> in <module>()
----> 1 sin("five")


TypeError: Not implemented for this type
```

This happens when you try to use a Python function on the wrong data type. Here we've asked python to calculate the sin of the word "five", which is a string. Not unreasonably, Python is unable to do this.

3.2.3 SyntaxError

```
In [23]: sin(4.654
```

```
  File "<ipython-input-23-f62c1c0f182e>", line 1
sin(4.654
        ^
SyntaxError: unexpected EOF while parsing
```

A syntax error means you've made a mistake with the combination of characters Python was expecting. This is sometimes due to a simple typo, and very frequently due to mismatching parentheses. In this example it's obvious that we've missed out the closing bracket, but in a complicated expression with lots of nested brackets, it's easy to lose track (and sometimes frustrating to try and work out where the missing bracket needs to go!)

3.3 Strategies for getting your code to work

The best strategy for getting your code to work properly is to follow guidelines for good programming practice. We'll add to this list as we go along, but for the moment:

- Write clear code that is easy for humans to read
- Use intuitive variable names
- Include comments so that anyone reading your code will know what you think your code is doing
- Make sure all the output of your code is labelled (including appropriate units) so that the user knows what it means.

---

4. More structured code

So far we've only considered linear codes - we start with the first line of the program, executes each line in sequence until we get to the end. More powerful code is able to do different things, or do the same thing repeatedly, depending on different conditions that we can set. We'll end this session by looking at comparison operators, and the "if" statement in Python.

---

5. Comparison operators

Conditional statements allow us to control the flow of a program. For example, consider this statement:

"If it's raining outside, then take an umbrella with you, else take your sunglasses"

Breaking this structure down, we can see that is has the form:

If *some condition* is true **then** *do something*, **else** *do something else*

Putting our statement in this form gives:

If "raining" is true **then** take umbrella, **else** take sunglasses

Python has a very similar structure for conditionals, but before we look at this in detail, let's look at how we test whether or not the condition is met.

For this, we use comparison objects, which once more work in a similar way to the way they do in maths. In Python these operators are written, fairly intuitively, as:

- == Is equal to
- != is not equal to
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

Each of these comparison operators can return one of two answers:

- True
- False

(Note the initial capitalisations)

For example:

```
In [24]: x = 10
         y = 3.5

In [25]: y <= 2.3 # is y less than 2.3?

Out[25]: False

In [26]: y > 4 # is y greater than 4?

Out[26]: False

In [27]: x == 10 # is x equal to 10?

Out[27]: True

In [28]: x + y != 3 # is the sum of x and y not-equal-to 10?

Out[28]: True
```

If we've got the hang of this, then we can use this to decide whether or not to do something else, using an "if" structure.

5.1 "If" structures

Here's an code snippet using a conditional statement in Python. In this example the variable x is an integer. You can see that it has the same basic structure as the umbrella example above:

if [something is true]:

- do this

else:

- do that

```
In [29]: # set an integer value for x. Try changing
         # this value and re-running the code
         x = 13
         if (x/2) * 2 == x:
             print " x is even."
         else:
             print " x is odd."
```

```
x is odd.
```

Note the format:

- The lines beginning "if" and "else" end with a colon :
- The lines following them are indented by four spaces - most code editors will do this for you automatically.

- Unlike some other languages, there is no "end if" statement. Python will treat all indented code as part of the if structure, when the indentation ends, the structure ends.

Note also the comparison operation:

- There's an integer division term, which will return an integer (and discard any remainder if x is odd)
- Thus when we multiply this integer by 2, we will obtain either:
  - the integer we started with, x. The comparison operator returns "True", and therefore x must be even.
  - an integer one less than x. The comparison operator then returns "False", and so x must be odd.