IOC Topic 7A.2 Part 3 – Advanced Python

Transcript & Notes

Author: Dr. Robert Lyon

Contact: robert.lyon@edgehill.ac.uk (www.scienceguyrob.com)

Institution: Edge Hill University

Version: 1.0

Topic 7A.2 Part 3, Introduction Slide

Hello and welcome to Part 3 of Topic 7A, Module 2, Advanced Python. In this module we aim to provide some background that will help you understand the programming you've already done. We eventually build upon this background and introduce some of the "advanced" features of Python. We'll also explore the Object-Orientated approach to software design. My name is Dr. Robert Lyon, and I'll be taking you through this module.

Slide 1

In Part 3 we will reinforce our understanding of Python by going over,

- Variables
- Keywords
- Objects
- Sets
- Other data structures
- Functions.

The aim: to get you back programming in Python outside of code academy, acquiring new experience along the way.

Slide 2

During part 3 we'll hop between the slides and an interactive coding environment called Google Collaboratory.

- Google Collaboratory is online environment that contains all the tools necessary to write and execute Python programs.
- To use the Collaboratory, all you'll need is a google account, e.g. a Gmail Account and the Chrome Web Browser.
- When you login to the Collaboratory, it creates a computer just for you to work with.
- The environment will allow you to create Python code and execute it.

- The environment is special, as it allows code to be executed in an interactive fashion. This means you don't need to write a whole source code file, before you can run some code.
- Instead you can execute individual commands one at a time. This is great for learning.
- Before proceeding, please create a google account if you don't already have one.

Slide 3

- To open the Colab environment, open search for it via Google.
- Once you've found it, click on the link. This will cause the Colab to start loading. It will take a few seconds, after all, its creating an entire Python environment for you to use.
- We can see the default page has some useful information and tutorials. I strongly suggest you read through these to understand the environment before proceeding. For those who are impatient, like me, I move swiftly on to opening my own Colab "Notebook".
- This creates an empty notebook with not much happening inside of it. You can see there are some option menus, and a single grey "cell" at the top of the page. The name of the file that has been created for this Colab session is called "Unititled2.ipynb". Don't worry about what the file extension means for now.
- The cell at the top is a "code" cell. This means we can write code into it, and have it executed directly by the Python interpreter. Here I simply print out a string and execute it by clicking on the play button, to the left of the cell. This takes a few seconds to execute. Any code that is inside a code cell can be executed.
- You can create text cells in the Colab too. You can write plain text into these, and such cells will not be executed as code. You can hop back into text cells, or any cells for that matter, by double clicking on them. I do this here to correct my mistake.
- Next, I try to do some basic math to show you that the code is really being executed in the environment.
- Now I add another text cell. You can mix text and code cells any way you like. You can also add rich text formatting to text cells, as I show you here.

Slide 4

- Since you now have an idea of what the Google Colab environment is all about, I can direct you to a resource I've created for this module. This resource is a Colab notebook (a Jupyter notebook, actually) that contains code, hints, tips, links to videos and tutorials that you'll find useful. The notebook includes instructions on how to use it, so please read through those carefully. The main thing to remember is that you need to create your own version of this notebook to be able to run and edit it. This too is explained in the notebook.
- Remember you need to use Google Chrome to access the Colab environment.
- The link to the resource is shown on the slide: https://colab.research.google.com/drive/1JNwsQ6PM7IifWK2fXMEa0PPjw2RjxIMj .
- Head over to the notebook whenever you're ready. The notebook will direct you to return to the slides, so be sure to follow that guidance to make the transition between slides and practical activities as seamless as possible.

Slide 5

- So far, we've used code academy to write our code.
- However, we've not thought about the best way to write code – how to make it clear, concise, and easy to maintain.
- There is a set of standards, some might say guidelines, for writing "good" Python code.
- The standard was created when a Python Enhancement Proposal (PEP) was approved.
- PEP 8 defined some coding standards which I strongly encourage you to adopt.
- The code we write during this module will adhere to this standard – but if you're ever in doubt about coding styles or conventions – check the standard.
- Good programmers try to keep to the standards!
- A link to the PEP8 coding standards is provided on the slide: https://www.python.org/dev/peps/pep-0008/#comments .

Slide 6

Now we're hopping straight back into Python programming. Let's review writing comments in Python.

Why do we comment our code?

- To explain it works.
- To make it easier to read.
- To help make the code easier to maintain, for those who didn't write it.
- It is important to write informative, succinct comments.
- There are two types of comment:
    - In-line comments
    - And Block comments

Slide 7

- The hash (#) symbol is used to start a comment. The python interpreter ignores any text following the hash symbol in code.
- An in-line comment explains a single line of code. It may explain the purpose of the line or provide important information and pointers. Such comments should be used only when required. In the notebook I've provided for you, I use them a lot – mainly because I'm trying to explain things. I wouldn't use so many comments in most software I write.
- A block comment explains one or more lines of code. Blocks can spread over multiple lines if required, depending on the complexity of the code. If a comment needs more than one paragraph, split the comment up using an empty comment line.
- Head over to the notebook, to explore writing comments.

Slide 8

We've already created all sorts of variables.

- Numerical variables such as height = 1.86.

- String variables such as name = “Rob”.
- Boolean variables such as like_programming = True.
- Variables are stored in memory. The Python interpreter takes care of exactly where in memory that is.
- The Python interpreter also figures out what type variables are for us. It is up to us as programmers to ensure we mix the types in ways that aren’t intended. For example,
    - If we have the numerical variable a = 1.
    - Then if we run a command that tries to add some text to that number, we’ll encounter an error: print(a + “some text”) # produces an error!

Slide 9

- We’ve encountered four different data types so far.
- Head over to the Colab notebook, to explore these variables for yourself.

Slide 10

- We must be careful when using variables – we must use them together appropriately.
- We can convert some variables to other types. For example, we can convert the text value “1” to a numerical value of 1. To do this we “cast” the variable to a different data type.
- We can perform this casting successfully, if we understand a little more about numerical variables - they may be,
- Integers – whole numbers on the number line such as … -2, -1, 0, 1, 2 … and so on. We can see some integers represented by the number line below. The number line is infinite, thus there are an infinite number of possible integers.
- Floating point numbers – numbers with fractional components such as … -2.1, -1.3, 0.02, 1.01, 2.4 … and so on. There are infinitely many floating-point numbers too.

Slide 11

- We can convert any integer to the floating-point numerical data type via the command float(). For example:
- If we have the integer variable: a = 1.
- We can cast it to a float inside of a print statement like so: print(float(a))
- We can also convert a text string containing only an integer, to an integer type via the command int(). For example:
- If we have the string variable a = “1”.
- We can cast it to an integer inside of a print statement similarly: print(int(a))
- **We** can convert an integer or float to string type too, via the command str(). For example:
- **If** we have the integer variable: a = 1.
- **We** can cast it to a string inside of a print statement like so: print(str(a)).

Slide 12

- We can convert a float to an integer via the command int(). For example:
- Let's say we have a variable called: a = 1.33
- We can cast that variable to an integer within a print function as follows: print(int(a))
- However, the output is: 1
- What's happened here? Well, integers don't have fractional parts. So where does the .33 go? In Python and all other computer programming languages, this fractional component gets truncated (or chopped off!).
- When converting a float to an integer, the float is either rounded up or down to the nearest whole number, depending on the programming language.
- In Python floats are rounded down. Try running this:
- Set a float equal to 2.9: a = 2.9
- Then cast that variable to an integer within a print function as follows: print(int(a))
- Head over to the notebook, to explore creating and casting variables for yourself.

Slide 13

- String variables contain textual information.
- Strings are present in lots of programming languages, including Python.
- I realize you've already created strings in code academy, for example:
    - Here's a string variable called name, that contains the text "Rob": name = "Rob"
    - We know we can print out string variables with ease too: print(name)
- You also know about escape characters you can insert into strings to improve the formatting. For example,
    - The newline character: "\n".
    - The carriage return character: "\r".
    - The tab character: "\t".

Slide 14

- String variables can be used in different ways. You know you can print them: print("A string")
- You know that you can access individual characters in the string using the following notation:

```
text = "Hello"

print(text[0]) # Prints out 'H'
```

- You also know you can access whole parts of strings using "slicing". We can do this with the following notation: print(text[0:2]) # Prints out 'He'
- Head over to the notebook to explore strings a little more.

Slide 15

- In Python, strings are immutable. This means that once they're created, they can't be changed.
- This may seem strange, as you probably feel like you've changed strings in your code.
- In reality all strings you create are put in memory and do not change.

- It's important we know this, otherwise we might introduce errors into our code.
- Actually, many Python objects are immutable – but there are exceptions. We can see here that booleans, integers, floats, strings and the frozen set class are immutable. We look at the mutable objects in upcoming slides.
- For now, Head over to the notebook to explore immutability.

Slide 16

- All computer programming languages have some way of indicating where a line, or block of code, finishes. For instance, here we can see that in the Java programming language, sections of code are grouped using brackets and lines are terminated with semi-colons. Code within the brackets is run together line after line, whilst code outside is run separately.
- In Python, blocks of code are defined using indentation. Here we can see Python code that does exactly the same thing as the Java code. Yet there are no braces or semi-colons. The Python interpreter treats all text on a line as potential code (excluding comments), so we don't need the semi-colon. To separate the code within the loop from code outside, all code within the while loop is indented using space characters. This indentation is done relative to the definition of the while loop.
- When writing your Python code, you may have encountered problems due to how your code was indented – maybe even errors.
- Most students new to Python struggle with indentation. But even experience programmers can make indentation mistakes – ALL programmers make mistakes! As for which style is better – braces and semi-colons versus indentation… well that's mostly a matter of personal preference. Some languages do require less syntax and therefore keypresses when coding. This becomes important when writing millions of lines of code!

Slide 17

- The PEP8 standard tells us to use 4 spaces per indentation level. If we look back at our while loop, we can show this visually.
- As we progress through the slides, we'll see how indentation is used – try your best to adopt the correct approach.
- You can head over to the notebook to explore indentation.

Slide 18

Python lists are mutable collections. Suppose we have,

list = ['r', 'o', 'b', '!']

As they are mutable, we can change them without creating new objects. For example, if we do,

list[3] = '.'

then we change the list directly. If we do this with a string, e.g.

name = "Rob!"

Then change the character at position three of the string to a full stop:

name[3] = '.'

We get a new string in memory.

Slide 19

- There's so much more you can do with lists. You can,
    - Add items to lists.
    - Delete items from lists.
    - Add lists together.
    - Use in-built functions to find the minimum or maximum elements in a list.
    - Count how many times an item occurs in a list.
- At this point head over to the notebook and try some lists exercises.

Slide 20

- Python dictionaries are also mutable collections.
- They contain pairs of keys and values. Each key unique identifies a set of values in the dictionary. For example, here we have an integer key of 1, which points to a collection of string values. The keys don't have to be strings, they could be some other type. The values could also be multiple strings.

We can create dictionaries with simple commands:

dict = {"1": "Rob!"}

We can change the dictionary directly:

dict["1"] = "Rob."

Or access dictionary elements:

print(dict["1"])

Head over to the notebook to learn more about using dictionaries.

Slide 21

There are many operators in Python. You might know some of them.

- **There** are logical operators such **as**: not, and, or.
- **Equality** operators such **as**: is, is not, exactly equal to, not equal to.
- **Comparison** operators such **as**: less than (<), less than or equal to (<=), greater than (>) and greater than or equal to (>=).
- **Finally,** there are arithmetic operators: Addition +, subtraction - , multiplication *, and division /.

Slide 22

To become advanced Python programmers, we need to understand how these operators relate to one another, and how they are used to evaluate expressions. An expression can be as simple as:

x = 4 * 1 # Which equals 4.

What about more complex expressions – what's the value of x here?

x = 4 + 9 -1 * 3 / 6 # It equals 12.5.

The answer is 12.5. Do you understand why this is the case? It's because division and multiplication have higher precedence. Here's what happened in steps:

1. Step 1. 3 / 6 = 0.5 (division first)
2. Step 2. -1 * 0.5 = -0.5 (multiplication second)
3. Finally, step 3. 4 + 9 – 0.5 = 12.5 (addition and subtraction last).

Slide 23

- It's important to use operators correctly, otherwise you'll get unexpected outputs. We can use brackets to force expressions to be evaluated in the order we want, for example, here we can see the previous expression evaluated a little differently.

  x = ((4 + 9 - 1) * 3) / 6 # Equals 6

- Here's a table showing the precedence of some, but not all operators.
- Head over to the notebook to examine precedence further.

Slide 24

- It's important to use operators correctly, otherwise you'll get unexpected outputs. We can use brackets to force expressions to be evaluated in the order we want, for example, here we can see the previous expression evaluated a little differently.

  x = ((4 + 9 - 1) * 3) / 6 # Equals 6

- Here's a table showing the precedence of some, but not all operators.
- Head over to the notebook to examine precedence further.

Slide 25

- Often, we need to control the flow of our Python programs, based on one or more conditions.
- In Python, If-else statements allow us to this.
- These statements evaluate one or more variables in terms of,
  - Equality (are the variables equal).
  - Whether or not their values fall in some specific range.
- You can have multiple tests in an If-else statement.
- Sometimes we may want to check for more than one condition. Python provides us with the 'elif' keyword for just this scenario.
- We can therefore build up complex control-flow code using these statements.
- More examples are provided in the notebook, so head over there now.

Slide 26

- It is often desirable to repeat a piece of code, until some condition is met. In Python this can be achieved with a while loop.
- For example, while the user is yet to pick a file, wait for them to choose.
- We can visualise what while loops are actually doing to better understand them:
  1. The code reaches some point represented by the blue circle.
  2. A pre-defined condition is then checked by the while loop.
  3. If the condition does not hold, execution skips past the while loop and carries on.
  4. If the condition does hold, whatever code is in the code block is repeated.
  5. The code returns to the start point represent by the blue circle, and the process repeats again.

Slide 27

While loops can be declared as follows:

```
condition = True
```

While condition:

```
print("This condition remains true…")
```

- If the variable condition is never set to False, this statement will loop forever!
- This isn't good, so always ensure your conditions will be updated properly when using while loops.
- Head over to the notebook to try some more examples.

Slide 28

- For loops are similar to while loops, however they give you access to variables capable of maintaining counts, useful for many tasks.
- To make such a for loop, we use a Python standard library function, range(), to keep count.
- For example, this code will print out the numbers 1 to 9:

```
for num in range(1,10):

print(num)
```

If we want to include the number 10, we must modify our inputs to the range function like so.

```
for num in range(1,11):

print(num)
```

Get use to using the range function, it is very useful:

https://docs.python.org/3/library/functions.html#func-range

Slide 29

- Python also supports for loops, capable of iterating over collections, such as lists.
- Here's a simple example that iterates over a collection of strings:

```
x = ["a", "b", "c"]
```

```
for text in x:
print(text)
```

- Which will print out:

a

b

c

Head over to the notebook for more examples and activities.

Slide 30

- Functions are reusable self-contained units of code that are incredibly useful.
- Functions have a standard structure in Python.
- They have a function signature, which defines their name and their usage. In this example, we have a function called my_function, that accepts two inputs.

```
def my_function(input_1, input_2):
    answer = input_1 + input_2
    print (answer)
    return answer
```

- Then we have the function body, which contains the code executed by the function. Any variables declared within this function, remaining within the "scope" of this function. That is, they aren't available outside the function, unless returned by the function and stored. Here the function returns a variable called answer.
- The function above takes two variables, adds them together, then returns the value.
- Functions can accept multiple input parameters.
- Functions can return a value, but they don't have to.

Slide 31

- In the last slide "scope" was mentioned.
- Scope refers to the area of the code that variables or functions are available to be used.
- If a variable is defined in a function, for example, it's scope is confined to the function.
- This means it can only be seen within the function, and not elsewhere.
- It also means that variables defined outside of functions, can't be seen or edited within the functions.
- These issues trip up novice programmers, so we'll explore scope for ourselves now – head over to the notebook and find the section on scope.
- Scope is a difficult topic to explain using slides, but the notebook will help us to learn via examples.

Slide 32

Here we've reviewed,

- Python coding standards.
- Basic variables.
- Casting.
- Strings.
- Immutability.
- Indentation.
- Lists and Dictionaries.
- Operator precedence.
- If else statements, while loops and for loops.
- Functions.
- Variable scope.

That’s an awful lot of material! In part 4, we’ll encounter some more advanced topics.