MATH6005 Introduction to Python. Lab 1

January 23, 2019

1 Python Basics

1.1 Introduction

Welcome to the labs for MATH6005, Introduction to Python. This lab work assumes you are using `spyder 3` and `Python 3.x`. You might find that the University labs have both spyder 2 and spyder 3 installed. **Please make sure you are using** `spyder 3`.

The lab consists of a series of examples, explanations and exercises. Solutions to exercises can be downloaded from blackboard.

- To help you with Python basics we have created a YouTube playlist
- https://www.youtube.com/watch?v=u0yDOsUYxZg&list=PLU2JUjGUsUm7R-3VdQA5S6IIaTK6YN1xE

1.2 Setup

Python 3 is already installed and ready to use in University computer rooms.

You may use your own laptop if you want. Python is free. Installation may take 10 - 15 minutes, so please *don't* do it right now (although you may want to download the installer on campus - it is not small).

- If you are installing Python we recommend installing it via Anaconda:
- https://www.youtube.com/watch?v=u0yDOsUYxZg

For the labs, you will need to open `spyder`. On the bench PC, go to the Start menu -> All Programs -> Programming Languages -> Anaconda3 (64 bit), and then choose `spyder`. You may try searching for it but there may be multiple versions installed: you need to ensure that the version you use has **Python version 3**, not version 2.

- We recommend that you view our introduction to Spyder video:
- https://www.youtube.com/watch?v=bb-Y5ylTAZM&index=2&list=PLU2JUjGUsUm7R-3VdQA5S6IIaTK6YN1xE

1.3 The print() function

One of the most useful functions in Python is the `print()` function. It is used to display information to the user. It can be used to present the result of computations, intermediate calculations, general text and used for debugging.

Once you have opened `spyder`, we will first look at the *console* in the bottom right part of the screen. That allows us to type in Python commands and get an *immediate* response. Let's use it to learn how to use `print()`

Let's use `print` to display a message on the console. Type in the following to console:

```
In [9]: print('hello world')

hello world
```

If we want to include the result of a computation in the output from print we use the following format:

```
print('1 + 1 = {0}'.format(1+1))
```

Try running the code above. The value in the curly brackets '{}' is replaced by the value 2 (i.e. 1+1).

`print()` can include the output from multiple computations if needed e.g.

```
print('1 + 1 = {0} and 2 + 2 = {1}'.format(1+1, 2+2))
```

The numeric value in the curly brakets corresponds to the index of value found in `.format(1+1, 2+2)` I.e. {0} is linked to 1+1 and {1} is linked to 2+2.

You don't have to use the indexes in acsending order, but if you don't do make sure that your code makes sense! e.g.

```
In [16]: print('1 + 1 = {1}!? and 2 + 2 = {0}!?'.format(1+1, 2+2))

1 + 1 = 4!? and 2 + 2 = 2!?
```

- You can also format your output to a specified number of decimal places using `{0:.2f}` instead of `{0}`
- The .2f after the : tells python that the number is a floating point and that you would like it shortened to 2dp.

```
In [18]: print('The number {0} given to 2 decimal places is {0:.2f}'.format(3.14159))

The number 3.14159 given to 2 decimal places is 3.14
```

- Similarly if you wanted to show the number to 3 decimal places you would use

```
In [20]: print('The number {0} given to 3 decimal places is {0:.3f}'.format(3.14159))

The number 3.14159 given to 3 decimal places is 3.142
```

1.4 Basic mathematics in the console

We have already seen that python can be used for basic mathematics when learning how to use `print()`

Let's use the iPython console as a calculator. Try the following calculations:

```
In [1]: 1 + 1

Out[1]: 2

In [2]: 1 / 2

Out[2]: 0.5

In [23]: (1 + 2.3 * 4.5) / 6.7

Out[23]: 1.6940298507462686
```

- the `**` operator raises one number to the power of another.

```
In [4]: 3**2

Out[4]: 9
```

As before we can mix basic mathematics with `print()` e.g.

(Note: if you are unfamiliar with the mod operator, it operates like a remainder function. if we type `15 % 4`, it will return the remainder after dividing 15 by 4.)

```
In [2]: print('Addition: {0}'.format(2+2))
        print('Substraction: {0}'.format(7-4))
        print('Multiplication: {0}'.format(2*5))
        print('Division: {0}'.format(10/2))
        print('Exponentiation: {0}'.format(3**2))
        print('Modulo: {0}'.format(15%4))

Addition: 4
Substraction: 3
Multiplication: 10
Division: 5.0
Exponentiation: 9
Modulo: 3
```

1.5 The Spyder editor

Using the console is fine, but has problems. We want to keep our work for future re-use. We want to systematically build up a lot of code. Neither of these is easy within the console. Instead, we can use the *editor* on the left half of the screen. This acts like a word-processor, allowing us to type in commands that we can then use or run later.

In the editor, remove any existing text and type in some of the commands you've previously used:

```
1 + 1
1 / 2
print("Hello {0}".format(1 + 2))
```

Save the file in a sensible location (your filestore, or the Desktop) under the name `lab1.py`.

We then want to run the commands in the file. To do this, choose "Run" from the Run menu, *or* press the big green play button on the toolbar, *or* press `F5`.

The output you see should look like:

```
In [26]: 1 + 1
         1 / 2
         print('1 + 1 = {0}'.format(1+1))

1 + 1 = 2
```

Notice that the output only shows the result of the print function. This shows one key difference between files and the console: only output that is *explicitly* printed appears on the screen.

1.5.1 Exercise 1: Calculate a factorial in the iPython console and editor

Explicitly compute $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$ in the console. Then do the same in the editor, printing it out with explanatory text.

1.6 Variables

We want to be able to store data and results of calculations in ways we can re-use. For this we define variables names.

- Variables names can only contain letters, numbers, and underscores. Spaces are not allowed in variable names, so we use underscores instead of spaces. For example, use `student_name` instead of `student name`.
- Variable names should be descriptive, without being too long. For example `mc_wheels` is better than just `wheels`, and `number_of_wheels_on_a_motorycle`.

Here is an example variable called `salary`

```
In [33]: salary = 30000
         print(salary)
         print('The value of the variable called salary is {0}'.format(salary))

30000
The value of the variable called salary is 30000
```

A variable name is a **label** that's points to a location in your computer's memory.

In our example above, think of the variable as a post-it note with `salary` written on it. This points to the integer `30000` in the computer's memory. Then, when asked to `print` the value of `salary`, it `prints` the value with that label on it.

In Python we can move that label to any other variable of any type e.g.

```
In [41]: number = 1.2
         print(number)
         number = 'One point two'
         print(number)
```

```
1.2
One point two
```

There are certain rules and conventions for variable names:

- always start with a letter;
- only use lower case Latin letters, or numbers, or underscores;
- in particular, never use spaces or hyphens (which can be interpreted as a new variable or a minus sign respectively).

Mathematical functions also work on variables.

```
In [37]: salary = 30000
         tax_rate = 0.2
         salary_after_tax = salary * (1 - tax_rate)

         print(salary_after_tax)
```

```
24000.0
```

- Each variable has a **data type**.
  - We can check the data type of a variable using the built-in function `type`
  - For example, `salary` has the data type `int` (short for integer)
  - and `salary_after_tax` is of type `float` (short for floating point number)

```
In [42]: salary = 30000
         tax_rate = 0.2
         salary_after_tax = salary * (1 - tax_rate)

         print(type(salary))
         print(type(salary_after_tax))
```

```
<class 'int'>
<class 'float'>
```

- Notice that we didn't need to tell (or declare to) Python the data type of each variable
- This is because Python is a **dynamically typed** programming language.
- Python infers the type of a variable at runtime (when the code is run)
- The most common primitive data types in python are:

```
In [45]: foo = True    # bool (Boolean)
         bar = False   # bool (Boolean)
         spam = 3.142  # float (floating point)
         eggs = 10000000 # int (integer)
         foobar = 'elderberrys' # str (string)

         print(type(foo))
         print(type(bar))
         print(type(spam))
         print(type(eggs))
         print(type(foobar))
```

```
<class 'bool'>
<class 'bool'>
<class 'float'>
<class 'int'>
<class 'str'>
```

1.7 Strings

We have already used strings extensively.

Strings are sets of characters. Strings are easier to understand by looking at some examples. Strings are contained by either single, double quotes or triple quotes.

```
In [5]: my_string = "This is a double-quoted string"
        my_string = 'This is a single-quoted string'
```

Double quotes lets us make strings that contain quotations

```
In [46]: quote = "Jack Reacher said, 'Hope for the best, plan for the worst'"
         print(quote)
```

```
Jack Reacher said, 'Hope for the best, plan for the worst'
```

```
In [49]: multi_line_string = '''triple quotes let us split strings
         over mulitple lines'''

         print(multi_line_string)
```

```
triple quotes let us split strings
over mulitple lines
```

1.7.1 Exercise 2: Creating and using variables

A rectangular box has width 2, height 3, and depth 2.

Task: * Create a variable for width, height and depth. * Compute the volume of the box, assigning that to a fourth variable. * Print the result along with formattted explanatory text.

1.8 Comments in code

Comments allow you to write in you native language (e.g. English), within your program. In Python, any line that starts with a hash (#) symbol is ignored by the Python interpreter.

```
In [1]: # This an inline comment.
        print("This line is not a comment, it is code.")

        print("Python will ignore comments")  #comments can appear after code
```

```
This line is not a comment, it is code.
Python will ignore comments
```

What makes a good comment?

- It is short and to the point, but a complete thought. Most comments should be written in complete sentences.
- It explains your thinking, so that when you return to the code later you will understand how you were approaching the problem.
- It explains your thinking, so that others who work with your code will understand your overall approach to a problem.
- It explains particularly difficult sections of code in detail.

1.9 Functions and import

We won't get very far with just basic algebraic operations. We'll want to perform more complex computations. For that we need python functions.

Python has built-in mathematical functions. For example, * abs() * round() * max() * min() * sum()

These functions all act as you would expect, given their names. Calling `abs()` on a number will return its absolute value. The `round()` function will round a number to specified number of decimal points (the default is 0).

Additional functionality can be added in with using various packages such as `math` or `numpy`. We will explore `numpy` in more detail later in the course.

To use these packages you need to first `import` them into your code.

```
In [110]: import math
```

The math library adds a long list of new mathematical functions to Python. It is documentated here: https://docs.python.org/3.7/library/math.html

```
In [111]: print('pi: {0}'.format(math.pi))
          print("Euler's Constant: {0}".format(math.e))
```

```
pi: 3.141592653589793
Euler's Constant: 2.718281828459045
```

Python's Math module includes some mathematical constants as seen above as well as commonly used mathematical functions.

```
In [112]: print('Cosine of pi: {0}'.format(math.cos(math.pi)))

Cosine of pi: -1.0
```

We can import specific constants and functions from python modules

```
In [5]: from math import pi, cos

        print('Cosine of pi: {0}'.format(cos(pi)))

Cosine of pi: -1.0
```

1.9.1 Exercise 3: Use a function to calculate a factorial

- Import the `math` library.
- If required use `help(math)` or https://docs.python.org/3.7/library/math.html to explore the math module
- Use `help(math.factorial)` to explore how you use the math factorial function.

- use `math.factorial()` to check your calculation of 6!

1.10 Defining Python Functions

So far we have used functions built-in to python such as `print()` and `math.cos()`

You will also need to define your own functions in Python.

1.10.1 Functions. Example 1: adding two numbers together

```
In [7]: def my_add(a, b):
            """
            Returns the sum of two numeric values

            Keyword arguments:
            a -- first number
            b -- second number
            """
            return a + b
```

The Python keyword to *define* a function is `def`. Each function has a name: in this case `my_add`.

The keyword arguments to the function are then a comma-separated list between round brackets `()`. This is in the same format as calling the function, but we are inventing the variable names to refer to the input within the function. So, however the user calls the function, the first argument that is passed in will be assigned the label `a` within the function itself.

Finally there is a colon `:` to end the line. That says that whatever follows is the content, or body, of the function: the lines that will be executed when the function is called. All lines within

the function to be executed **must** then be indented by four spaces. `spyder` should start doing this automatically.

The three quotes are *documentation* for the function: they have no effect. However, **any undocumented function is broken**. We can see the documentation by using the help function:

```
In [8]: help(my_add)

Help on function my_add in module __main__:

my_add(a, b)
    Returns the sum of two numeric values

    Keyword arguments:
    a -- first number
    b -- second number
```

We then include all the commands with the function that we want to run each time the function is called. Once we have a result that we want to send back to the place that called the function, we `return` it: this send back the appropriate value(s).

We can now call our function:

```
In [10]: print(my_add(1, 1))
         print(my_add(1.2, 4.5))

2
5.7
```

1.10.2 Functions: Example 2: Implementing a formula

Suppose that you are promised a payment of £2000 in 5 years time.

Assuming a compound interest *rate* of 3.5% what is the **present value (PV)** of this future value (FV)?

- We can calculate this with the forumla: PV = FV / (1 + rate)^n
- We do not want type the code for this calculation each time we need it.

- Instead we create a reusable function that we can call to do this for different FV, rate and n.
- The code is below. The function follows the same basic pattern as the simple `my_add` function.

```
In [37]: def pv(future_value, rate, n):
             '''
             Discount a value at defined rate n time periods into the future.

             Forumula:
             PV = FV / (1 + r)^n
             Where
```

```
        FV = future value
        r = the comparator (interest) rate
        n = number of years in the future

        Keyword arguments:
        future value -- the value to discount
        rate -- the rate at which to do the discounting
        n -- the number of time periods into the future
        '''
        return future_value / (1 + rate)**n
```

```
In [44]: #Test case 1
         future_value = 2000
         rate = 0.035
         years = 5
         result = pv(future_value, rate, years)

         msg = 'Using an interest rate of {0}, a payment of £{1:.2f}' \
               ' in {2} years time is worth £{3:.2f} today'

         print(msg.format(rate,future_value, years,result))

         #Test case 2
         future_value = 350
         rate = 0.01
         years = 10
         result = pv(future_value, rate, years)


         print(msg.format(rate,future_value, years,result))
```

```
Using an interest rate of 0.035, a payment of £2000.00 in 5 years time is worth £1683.95 today
Using an interest rate of 0.01, a payment of £350.00 in 10 years time is worth £316.85 today
```

1.10.3 Exercise 4: Write a function to convert fahrenheit to celsius

Open Spyder and use the code editor to do the following:

Define a function `convert_fahrenheit_to_celsius` that converts degrees fahrenheit to degrees celsius. The function should have a keyword argument for temperature in degrees fahrenheit and `return` a numeric value for temperature in degrees celsius.

Store the answer in a variable and then print the answer to the user. Answers should be shown to **2 decimal places**.

Conversion formula:

```
deg_celsius = (deg_fahrenheit - 32) / (9.0 / 5.0)
```

Test data

1. Fahrenheit = 20; Celsius = -6.67
2. Fahrenheit = 100; Celsius = 37.78

1.10.4 Exercise 5: Write a function to calculate velocity

- Define a function that calculates and returns velocity (metres per second).
- The function should accept two parameters: distance travelled (metres) and time (seconds)

```
velocity (m/s) = metres travelled (m) / time taken (s)
```

Test data

1. distance travelled = 10m; time taken = 5s. (Velocity = 2.00 m/s)
2. distance travelled = 100m; time taken = 0.12s. (Velocity = 833.33m/s 2dp)

1.10.5 Creating your own Python modules and importing functions

In the same way we imported functions from `math` we can import functions from our own python modules

- Open `py_finance.py` and `test_finance.py`
- `test_finance.py` imports functions from the `py_finance` module
- Watch the Youtube video that explains how they work:
- https://www.youtube.com/watch?v=l0qE5_01bzw&t=170s

1.11 Lists

Variables are useful, but in nearly all real programming problems we need to store and manipulate **lots** of data in memory.

For example, if we were developing a music streaming service we might need to hold the list of song's on a album or an artists back catalog.

Or, if we were developing a software to manage the geographic routing of a fleet of delivery vehicles we might need to hold a matrix of travel distances between postcodes.

A Python `List` is a simple and flexible way to store lots of variables (of any type of data).

```
In [19]: foo = [0, 1, 2, 3]
         print(foo)
```

```
[0, 1, 2, 3]
```

The square brackets `[]` say that what follows will be a list: a collection of objects. The commas separate the different objects contained within the list.

In Python, a list can hold *anything*. For example:

```
In [20]: bar = [0, 1.2, "hello", [3, 4]]
         print(bar)
```

```
[0, 1.2, 'hello', [3, 4]]
```

This list holds an integer, a real number (or at least a floating point number), a string, and another list.

We can find the length of a list using `len`:

```
In [21]: print(len(foo))

4
```

To access individual elements of a list, use square brackets again. The elements are ordered left-to-right, and the first element has number `0`:

```
In [22]: print(foo[0])
         print(foo[3])
         print(bar[1])
         print(bar[3])

0
3
1.2
[3, 4]
```

If we try to access an element that isn't in the list we get an error:

```
In [23]: print(foo[4])


         ---------------------------------------------------------------------------

         IndexError                                Traceback (most recent call last)

         <ipython-input-23-f1badaafaf0e> in <module>()
     ----> 1 print(foo[4])


         IndexError: list index out of range
```

We can assign the value of elements of a list in the same way as any variable:

```
In [24]: foo[1] = 10
         print(foo)

[0, 10, 2, 3]
```

The number in brackets is called the index of the item. Because lists start at zero, the index of an item is always one less than its position in the list. So to get the second item in the list, we need to use an index of 1.

We can work with multiple elements of a list at once using *slicing* notation:

```
In [25]: print(foo)
         print(foo[0:2])
         print(foo[:2])
         print(foo[1:])
         print(foo[0:4:2])
         print(foo[::2])

[0, 10, 2, 3]
[0, 10]
[0, 10]
[10, 2, 3]
[0, 2]
[0, 2]
```

The notation `[start:end:step]` means to return the entries from the `start`, up to **but not including** the `end`, in steps of length `step`. If the `start` is not included (e.g. `[:2]`) it defaults to the start, i.e. `0`. If the `end` is not included (e.g. `[1:]`) it defaults to the end (i.e., `len(...)`). If the `step` is not included it defaults to `1`.

To get the last item in a list, no matter how long the list is, you can use an index of -1. This syntax also works for the second to last item, the third to last, and so forth. You can't use a negative number larger than the length of the list, however.

```
In [26]: print(foo[-1])
         print(foo[-2])
         print(foo[-1:0:-1])

3
2
[3, 2, 10]
```

If you want to find out the position of an element in a list, you can use the index() function. This method returns a ValueError if the requested item is not in the list.

```
In [27]: print(foo.index(10))

1
```

You can test whether an item is in a list using the "in" keyword. This will become more useful after learning how to use if-else statements.

```
In [28]: print(10 in foo)
```

```
True
```

```
In [29]: print(11 in foo)
```

```
False
```

We can add an item to a list using the append() method. This method adds the new item to the end of the list.

```
In [30]: foo.append(12)
         print(foo)
```

```
[0, 10, 2, 3, 12]
```

We can also insert items anywhere we want in a list, using the insert() function. We specify the position we want the item to have, and everything from that point on is shifted one position to the right. In other words, the index of every item after the new item is increased by one.

```
In [31]: foo.insert(1,13)
         print(foo)
```

```
[0, 13, 10, 2, 3, 12]
```

We can remove an item from a list using the `del` statement. You need to specify the index you wish to remove.

```
In [32]: del foo[2]
         print(foo)
```

```
[0, 13, 2, 3, 12]
```

1.11.1 Exercise 6: Marvel Comics

You are given a list of comics:

```
comics = ['Iron-man', 'Captain America', 'Spider-man', 'Thor', 'Deadpool']
```

Tasks:

- slice and then print the first and second list items
- slices and then print the second to fourth list items
- slice and then print the fourth and fifth list items
- append "Doctor Strange" to the list. Print the updated list
- insert "Headpool" before "Deadpool" in the list. Print the updated list
- delete "Iron-man". Print the updated list

1.11.2 Week 1: Debug Challenge

Each laboratory will have a debug challenge. You will be given a pre-existing script containing Python code. The catch is that the code doesn't run!

Your challenge is to find and correct the errors so that the script correctly executes.

The challenges are based around common problems students have when writing code. If you do the exercises it will help you debug your own code and maybe even avoid the mistakes in the first place!

Challenge 1: Instructions:

- open `week1_debug_challenge1.py` in `spyder 3`
- Attempt to run the code.
- Fix the bugs!

Hints:

- Read the Python interpreter output.
- The errors reported can look confusing at first, but read them carefully and they will point you to the lines of code with problems.
- The `Spyder` IDE may give you some hints about formatting errors
- It can be useful to use `print()` to display intermediate calculations and variable values.
- Remember that `Spyder` has a variable viewer where you can look at the value of all variables created.
- There might be multiple bugs! When you fix one and try to run the code you might find another!

Have a go **yourself** and then watch our approach:

- https://www.youtube.com/watch?v=XCuD59bYKx0

1.11.3 Week 1: Do something we haven't taught you challenge

Each laboratory will challenge you to do something that we haven't taught you before.

No course can teach you everything you need for all programming problems. Being a competent Python programmer means that you need to learn how to find solutions to problems yourself. These challenges are designed to help you begin to use internet resources in order to solve your problem.

Before you try the challenges it it worth watching our video on using StackOverflow:
https://www.youtube.com/watch?v=9WziNfkTRZ0&index=3&list=PLU2JUjGUsUm7R-3VdQA5S6IIaTK6YN1xE

Challenge 1: You are given a unsorted `List` of integers.

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

Find a command to sort the list into ascending order i.e.

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

Once you have tried **yourself**. Watch our example strategy: https://www.youtube.com/watch?v=369YdvOwrGU&list=PLU2JUjGUsUm7R-3VdQA5S6IIaTK6YN1xE&index=5

```
In [10]: unsorted = [5, 7, 6, 4, 3, 2, 1]
         print(unsorted)
```

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

Challenge 2: Sometimes a Python function needs to accept a variable number of arguments. For example, the built-in function `max()`

```
max(1, 2, 3)
max(1, 2, 3, 4, 5, 6, 7, 8)
```

Write a function that accepts a variable number of integer arguments and returns the number of arguments e.g.

```
result = number_of_arguments(1, 2, 3) # result = 3
result = number_of_arguments(1, 2, 3, 4, 5) # result = 5
```

Try to solve this yourself. Then watch our example strategy: https://www.youtube.com/watch?v=ClyHbrkpqJU&index=6&list=PLU2JUjGUsUm7R-3VdQA5S6IIaTK6YN1xE

Challenge 3: We have seen multiple Python functions that have required parameters. For example, the function `add_two(a, b)` **required** the user to provide two parameters `a` and `b`. It is also possible in Python to have **default values** for the parameters.

Write a function called `super_hero_name` that accepts two parameters of type string: `firstname` and `super_surname`.

The parameter `super_surname` should have a default value of 'the spider'. The function should concatonate the names and return the resulting superhero name.

E.g.

```
super_hero_name("tom") #returns "tom the spider"
super_hero_name("tom", "ant-man") #returns "tom ant man"
```

Try this yourself first. Then watch our approach: https://www.youtube.com/watch?v=mny4iKtT21s&index=7&list=PLU2JUjGUsUm7R-3VdQA5S6IIaTK6YN1xE

```
In [2]: def super_hero_name(firstname, super_surname):
            return firstname + ' ' + super_surname

        super_hero_name("tom", "the spider")
```

```
Out[2]: 'tom the spider'

In [3]: super_hero_name("tom", "ant-man")

Out[3]: 'tom ant-man'
```

1.12 Extra Material

Download and open `string_manipulation.py` for detailed examples of how to manipulate and format Python strings.