Python Practice 1: Getting started with Python

Preliminaries

The aim of this lab class is to get you started with programming in Python, and to introduce some of the resources available for the module. **Please take the time to read the material carefully**, and work through all the exercises.

Materials available from the module homepage

You will have accessed this sheet via the module homepage, by following the link to "Introductary Python Materials". This same sub-page provides access to *code and data files* that you will need when working with these python practice sheets, as well as some introductory notes (as slides) on Python.

The main module homepage also provides links to some instructions for using Python on a CiCS desktop.

Create a folder for your lab class files

You will need to work create many different code files as part of your lab class work. To store them, create a dedicated folder in the `U:` drive, e.g. named `COM6115`, etc. *DO NOT* use a folder on the desktop. You might also create separate subfolders for different labs. AVOID using names that contain any *spaces*, or non-standard characters, for yours codes files or any of the folders that contain them.

Working with the Python Interpreter

Begin by starting the `Spyder` editor/IDE, as follows: go to the search box (magnifying glass icon, next to START menu), and enter "`spyder`", then select the entry for the `Spyder desktop app`. You can use Spyder to create new code files, and can open an existing file, by opening its folder and "dragging-and-dropping" it from the folder onto Spyder. The *first time* you run a given code file, press "**ctrl + F6**" to configure how the code is run. Reasonable options include:

1. "**Execute in a new dedicated console**" *OR*
2. "**Execute in an external system terminal**" + "**Interact with the Python console after execution**" (Warning: this alternative will work ONLY IF there are NO SPACES in the code file name or PATH to the file on the computer.)

You can interact directly with Python an interpreter window (with prompt "`>>>`") on a Console window (with a prompt such as "`In[1]:`"). You can enter commands at the prompt that Python will execute, or expressions for it to evaluate. Some things for you to try follow. (Note that the examples below use the "`>>>`" prompt, rather than "`In[1]:`", but that is not important.)

PLEASE NOTE: *be warned!* — indentation is *very important* in Python. If a *single stray space* character precedes your expression, Python will not evaluate it (indicating an *Indentation Error*).

Arithmetic

Begin by entering some arithmetic expressions, using the basic arithmetic operators: `+` (plus), `-` (minus), `*` (multiply), `/` (divide). For example:

```
>>> 2 + 3
5
>>> 2.9 + 3.2
6.1
>>> (3.2 / 2.0) + 7
8.6
>>>
```

Try using different mixtures of the arithmetic operators, with and without brackets to fix the scope of the operations. For cases *without* brackets, see if you correctly anticipate the operator scopes that Python assumes. Try expressions that mix float and integer types.

Next, read sections 2.6 and 2.7 of the course textbook, to find out about the *rules of precedence* that Python uses to decide how to handle expressions involving multiple operators. Another useful source is: `www.mathsisfun.com/operation-order-bodmas.html`. Try entering some more expressions, to test your understanding (you will be tested on this later.)

Using variables

Enter some statements that use variables to store values. We use "`=`" to *assign* a value to a variable (whose name appears to the left of `=`). Elsewhere, when a variable is *evaluated* (e.g. when it appears in an arithmetic expression), its value is accessed and used at that position:

```
>>> x = 12.5
>>> x
12.5
>>> y = 3 * x + 2
>>> y
39.5
>>> x = x + 1.1
>>> x
13.6
>>>
```

Most of the time, it is better to use *meaningful* names for variables, rather than simple names such as `x`. For example, we might use a variable named `balance` to store our bank balance as a floating point number, or a variable `message` to store a string that is to be printed as a message. Using meaningful variable names makes your code much more readable and understandable, and is a key feature of good programming style. It not only helps someone else to understand your code (e.g. me, if I'm marking it), it also helps **you** understand your own code, which is vital if you're trying to figure out why it doesn't work, or does something that you're not expecting.

Note that variable names can contain letters, digits and underscores (i.e. `_`), but must begin with a letter. Note also that there are some special words – Python's *keywords* – that cannot be used as variable names. Read section 2.3 of the web textbook to find out about them.

Strings

Try entering some *strings*, which are just character sequences between *paired* quotation marks. Python doesn't mind if you use `"` or `'` for this, as long as you have the *same* quotation mark at both the start and end of the string, e.g.

```
>>> "Hello World!"
'Hello World!'
>>> 'Time for tea.'
'Time for tea.'
>>> "these quotes are not paired'
SyntaxError: EOL while scanning string literal
>>>
```

Some of the arithmetic operators, specifically `+` and `*`, have additional special meanings when used with strings. For example, if `s1` and `s2` are variables storing strings, we can evaluate expressions such as `s1 + s2` and `s1 * 10`. Try out these operations with the Python interpreter. Also, investigate *precedence* for these operators, e.g. by entering: `s1 + s2 * 10`

Simple printing

Try entering some simple print commands. The `print` command will print a single value, or multiple values that are separated by commas. You must have a pair of brackets around the values to be printed (as below). By default, `print` adds a space between the values, and a *newline* character at the end of what it prints. For example:

```
>>> greeting = "Hello Mark."
>>> print(greeting)
Hello Mark.
>>> balance = 123.45
>>> print(greeting, "Your balance is", balance, 'pounds.')
Hello Mark. Your balance is 123.45 pounds.
>>>
```

You can *prevent* Python from adding a *newline* at the end of what it prints by adding an extra (final) *keyword argument* `end`, which determines what is added at the end. Here, we can set this to be them 'empty string', as in e.g. `print('this','that',end='')`. You won't be able to see the difference when entering print commands to the interpreter, but the difference matters when you have multiple print statements in a code file.

Note that you can *add* extra linebreaks to what's printed by using the *special character* `\n`, for example try printing the string "`hello\n\nthere`".

Looking at our first Python program

Download the file `distance_convert.py` from the module homepage, and open it in Spyder. An easy way to do this is to *drag-and-drop* it onto the Spyder edit window. The file contains the following:

```
# Converts distance in miles to kilometers
miles = 22.7
kilometers = (miles * 8.0) / 5.0
print("Converting distance in miles to kilometers:")
print("Distance in miles:     ", miles)
print("Distance in kilometers:", kilometers)
```

The first line (preceded by `#`) is a *comment* — it is ignored by Python. The remaining lines are commands that we might have entered at the Python shell. If we call python to run this program (or *script*), it will execute each line in turn, producing results just as if they had been entered at the shell. Take a short while to study the code, and understand what it does.

Clearly, the code takes an initial distance in miles (i.e. the value assigned to the variables `miles`) and converts it to kilometers, on the basis that a kilometer is five eighths of a mile. The initial and converted distances are then printed, with some 'supporting text'.

Now ask Spyder to run the script. The *first time* you run a code file, press "**ctrl + F6**", to configure the execution - sensible options to choose are indicated earlier in this document (bottom of the first page). You can now click "Run" to execute your code. After this first run, you can just press **F5** to run the code again, under the same configuration.

Running this code produces the following results in the interpreter window:

```
>>>
Converting distance in miles to kilometers:
Distance in miles:      22.7
Distance in kilometers: 36.32
>>>
```

As it stands, we can only do other distance conversions by editing the script file, so that different initial values are assigned to the `miles` variable. Try doing this.

Taking input from the user

Python provides a function `input` for reading what a user types in whilst a program is running, which makes it possible to write programs that interact with the user. `input` takes a single (optional) argument, which is a 'prompt' string. When called, the function prints the 'prompt', and then reads in whatever text the user types up to the first line return. The function then 'gives back' (or *returns*) the value to the program. and below we assign it to a variable.

```
>>> s = input('Please respond: ')
Please respond:    this and that
>>> s
'   this and that'
>>>
```

Modifying the first Python program

To avoid having to edit the file `distance_convert.py` for each different calculation, we could instead use the `input` function to read in the start value for the calculation. Create a copy of the distance conversion program in a differently named file, e.g. "`distance_convert_v2.py`" (since it's generally poor practice to destroy your previous code), and modify the definition so that it uses the `input` function, prompting the user to enter a value for use in the calculation. We have an immediate problem, however, which is that `input` always returns a *string*, not a number. We can convert the string to a number by applying the `float` function, as in the following example:

```
>>> s = '3.45'
>>> s
'3.45'
>>> float(s)
3.45
>>> float(s) + 1
4.45
```

Writing our first Python program

Using the distance converter script as a template, write a new script for converting temperatures. *Create a new file* in your lab class folder for the script, i.e. *do not* overwrite the distance converter script file. As before, use Spyder as your editor. You might call your new script `temperature_convert.py` — it's up to you — but call it something sensible/informative.

Your script should do the following. Firstly, it should ask the user for a temperature in degrees celsius (a.k.a. *centigrade*). Next, it should compute the corresponding temperature values on both the *fahrenheit* and *kelvin* scales. It should then print a summary of its results, similar to that of the distance converter program.

As you probably know, to convert a celsius temperature to the fahrenheit scale, we multiply it by 9/5 and then add 32. The kelvin scale has the same sized units as the celsius scale, but its *zero* point corresponds to *absolute zero*, which is equivalent to $-273.15\,^{\circ}\mathrm{C}$, so converting from celsius to kelvin requires only that we add 273.15.

Defining a Python Function: reusable functionality

A good way of creating conveniently reusable chunks of functionality is to define a function. Converting the code of our original program `distance_convert.py` to have the form of a Python function might give the following:

```
def convert_distance(miles):
    kilometers = (miles * 8.0) / 5.0
    print("Converting distance in miles to kilometers:")
    print("Distance in miles:     ", miles)
    print("Distance in kilometers:", kilometers)
```

Here, the first line starts with the *keyword* `def`, indicating a function definition. What follows the keyword indicates that the function has name `convert_distance` and takes a single argument. When the function is called, the argument value provided is assigned to the variable `miles` (indicated in the brackets of the definition's first line). Note how the *body* of the definition is *indented* relative to the first line (which is not indented). This indentation is crucial — it signals that the indented lines *belong* to the body of the definition.

PLEASE NOTE: **be warned!** — getting indentation right is crucial to successful Python programming. You should *create* indentation by using the `TAB` key in Spyder. You should **never** attempt to indent code 'manually' by using *spaces*.

If you run this code, the interpreter will actually just read and remember the function definitions. We can then call the function as follows, i.e. using the function's name with the required argument value supplied in brackets.

```
>>> convert_distance(44)
Converting distance in miles to kilometers:
Distance in miles:      44
Distance in kilometers: 70.4
>>> convert_distance(122.5)
Converting distance in miles to kilometers:
Distance in miles:      122.5
Distance in kilometers: 196.0
```

Working in *a new file* (i.e. with a modified file name), modify your previous code to create *two alternative functions* for converting temperature — one converting degrees celcius to fahrenheit, the other from celcius to kelvin — with suitably distinct (and informative) function names.

Making your function return its values

A common mistake when learning Python is to confuse *printing* a value with *returning* the value. See the lecture slides (last slide on "Defining Functions" about the use of `return`). Modify your functions so that they *return* the value computed, i.e. so you can assign the value to a variable. For this you must add a use of the `return` command to your code. Note that when a `return` command is executed, it causes the function call to terminate, so it makes *no sense* to have any additional commands *after* the `return` command.

New programmers sometimes struggle to understand the difference between *printing* a value and *returning* it, but this distinction is **really important**. A key test is whether you are able to *assign* the returned value to a *variable*, so that you would be able to use it in another calculation. If you don't feel you understand this difference, then **ask the demonstrator**.

How to import your code

The usual way to access pre-existing code — either code you've written yourself, or code in library module — is to *import* it (i.e. as opposed to simply 'running' the code, as we did above, which doesn't usually make sense). For example, there is a module `math` providing many goodies, such as (e.g.) a variable `pi` defined with quite an accurate value of π. If we import this module with the statement `import math`, we would need to refer to this variable with the 'long name' `math.pi`. The alternative import statement `from math import *` brings everything (`*`) in with a 'local name', i.e. so we could refer to the variable as just `pi`.

Create a new file `test.py`, which imports from the code file containing your temperature conversion function definition, and then includes a series of test calls to this function. Note that, when importing from a local code file with a name such as `mycode.py`, the "`.py`" is dropped, i.e. we would use an import statement such as `from mycode import *`.

What else to do in this lab?

It's always a good idea to show your solutions to the exercises to a demonstrator. Their comments can be very helpful towards better coding in future.