INTPROG
Introduction to Programming

poolem.myweb.port.ac.uk/intprog

Practical Worksheet P01: Getting started with Python

Introduction

This worksheet is designed to get you started with the Python language and the Python programming environment, PyScripter. Work through it carefully at your own pace, making sure you understand each part before moving on to the next. In particular, when the worksheet asks you to type something in, first try to predict what it will do, type it in, and finally make sure you have understood what has happened.

You are not expected to finish worksheets in your practical session, but you should complete as much as you can before the practical of the following week — it is vital that you spend a few hours of your own time each week developing your programming skills. Use the practical session to ask questions on those exercises which you are finding difficult. If you are having major difficulties understanding the concepts, then recommended sources of help outside the lectures and practical sessions are: your fellow student programmers, the unit textbook (Zelle), the unit staff, and the Tutor Centre (Buckingham BK 1.16).

Start and Configure PyScripter

The programming environment that we'll use to write our Python programs is called PyScripter. Start PyScripter from MyApps. A window similar to that shown in the picture below will appear.

We can see that the PyScripter window includes three main regions. The area to the top-left is a file explorer. The large area to the right of this is an ***editor***, which we'll use later in this practical to write some programs. At the bottom is the Python ***shell***. The shell allows you to interact with Python directly. Each line you type into the shell is interpreted (executed) immediately, with any results displayed. This provides a good way to learn the basics of the Python language and to test out new concepts and ideas.

Before we use the shell, we'll change one of PyScripter's settings to make things less complicated to use. From the Tools menu, select Options, then IDE Options. In the first section (Code Completion) of the window that appears, make sure the boxes for "Complete as you type" and "Complete with work-break characters" are unchecked (and all the other boxes are checked). Click OK to save those settings permanently.

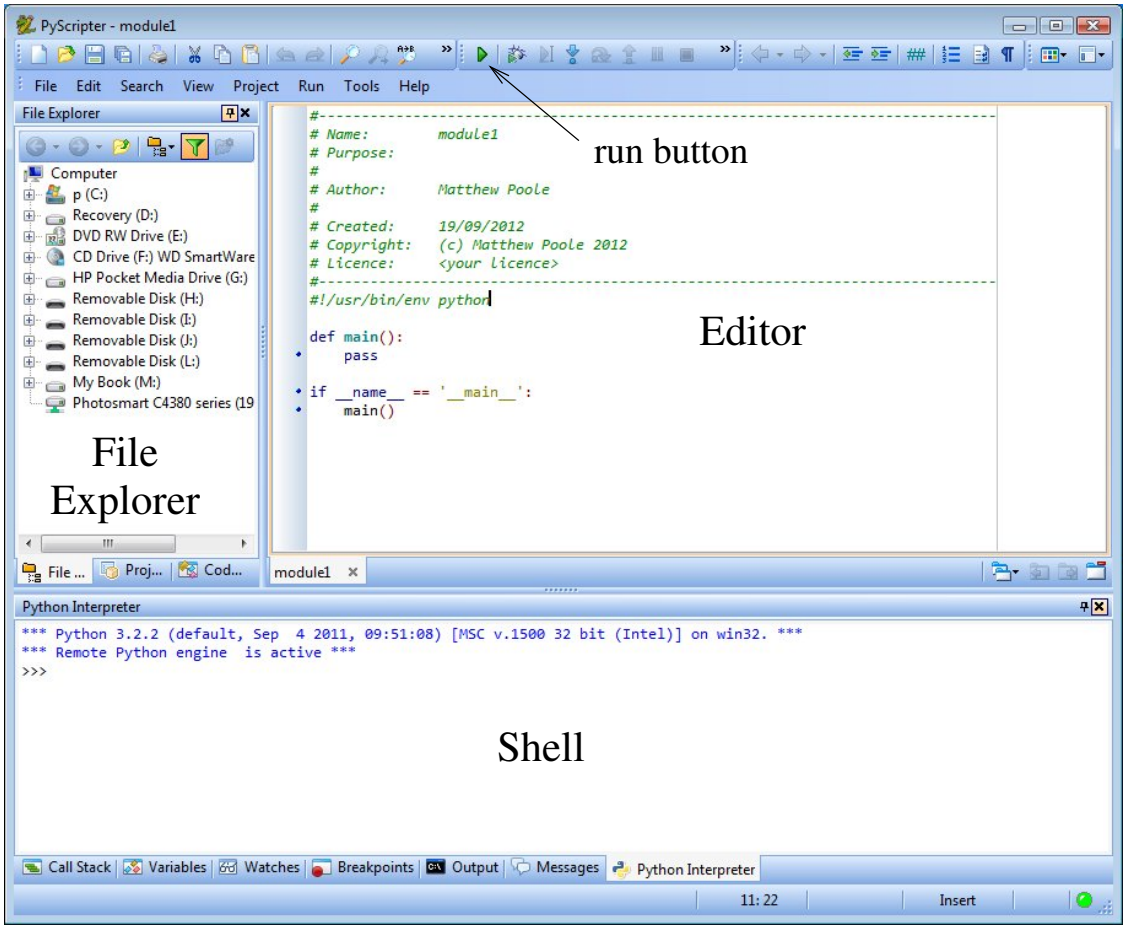
Your first Python program

Click the mouse within the shell, and at the shell ***prompt*** >>> type the following:

```
print("Hello world")
```

and press the return key. (Notice that when you type an open "(", PyScripter will automatically add a corresponding close ")" for you.) The shell executes the entered line, and once finished re-displays the >>> prompt ready for you to enter more code.

Note that each line of Python code is called a ***statement***, and this particular statement is known as a ***print statement***, since it uses the `print` function. This statement can also be considered to be a complete, albeit very simple, Python program!



Experimenting with the Python language

Let's continue using the prompt to experiment with the basics of the Python language. We have already seen above that we can use `print` to display text on the screen; `print` can also be used to display values of arithmetic ***expressions***. Try, for example:

```
print(7)
print(12 + 27)
print(5 + 4 * 3)
```

In fact, the shell will display the value of any expression that you type directly in at the prompt without using `print`; for example, enter: `5 + 4 * 3`. The shell can thus be used as a simple calculator.

Variables

Variables are basic elements of any program (in any programming language). Each variable has a name, and refers to a location in the computer's memory that holds a value. The value of a variable may change during execution of a program.

Let's experiment with some variables. When you start the shell there are no variables at all. To create a new variable, we use an ***assignment statement***; type:

```
x = 7
```

All assignment statements have this basic form: on the left of the `=` is a ***variable name*** (here `x`), and on the right hand side is an ***expression*** (here 7). This assignment statement means: "create a new variable `x`, and assign it the value 7". To check the value of your

new variable, type:

```
print(x)
```

or simply:

```
x
```

Let's create another variable `y`, using an assignment statement with a slightly more complicated expression:

```
y = x + 1
```

This assignment statement says: "create a new variable `y`, and assign it a value one greater than the current value of `x`". After entering this assignment, display the value of `y`.

Values of existing variables are changed using assignment statements. Try these statements, and inspect the values of `x` and `y` after each one:

```
x = x * 2
y = x + y
```

Experiment with variables and assignment statements until you are confident that you understand their behaviour. (By the way, a variable doesn't have to have a short name like `x` or `y` — just about any mix of letters and digits is fine, as long as it ***starts with a letter***.)

Shell history and code completion

A useful feature of the shell is that you can re-display previously executed statements by using the up and down cursor (arrow) keys. Once displayed, these statements can be modified and/or re-executed. Experiment with these shell history features.

Another useful time-saving feature of the shell is code completion, which means that you type the first few letters of a word, and by pressing ctrl+space, the shell attempts to complete the word for you. Try typing `pri`, holding down ctrl and pressing the space-bar. The shell has guessed that you want to write the word `print` and has completed it for you. Code completion will come in very useful later on when you will need to use much longer (multi-word) names in your code.

Errors

Like all programming languages, Python has very strict rules in terms of what you can ask it to interpret, and will report an error if you break these rules. ***Syntax errors*** are mistakes in the form (structure) of the code. They are often caused by typos and are usually easy to fix. For example, try the following code that contains syntax errors:

```
print(Hello World")
x = y + * 4
```

Semantic errors are mistakes in the meaning of the code. Try, for example:

```
print(2 * crash)
x = bang
```

These statements are syntactically OK (they look like valid print and assignment statements), but they don't make sense (assuming the variables `crash` and `bang` don't exist). After all, if a variable doesn't exist, it cannot have a value!

Note that many ***error messages*** (displayed in red) will be several lines long. Always look at the last line of the message: this may give you a clue as to what is wrong with your code.

Writing longer programs using the editor

Whilst the shell is great for experimenting with Python, it is only practical for short programs, and when we close the shell, we lose any program we've typed in! We need to be able to write longer programs, and to save them as files on the disk so that they can be executed over and over again. In order to do this, we will use PyScripter's ***editor***.

Begin by deleting all of the text within the editor and replacing it with your Hello World program:

```
print("Hello World")
```

Save it to disk using Save As from the File menu. Save it with filename hello.py into a suitable folder (I recommend creating a Intprog folder and storing all your Python files there).

Execute the program by pressing the ***green run button*** at the top of the window (or by pressing ctrl+F9). PyScripter executes the program within the shell. The program may be executed as many times as you like by pressing run again.

Now change the program so that it looks exactly as follows:

```
def sayHello():
    print("Hello World")
```

Execute the program again by pressing run.

This program will not give any output in the shell. Instead, it defines a new function. A function is a piece of code that has a name that we can use later. This particular function has the name `sayHello`. Now, try typing:

```
sayHello()
```

in the shell. Try it again. You should be able to see what is happening here — this `sayHello()` statement is a ***function call***, it uses (or "calls") the `sayHello` function we have just defined. Remember also to try out code completion here: type `say` and press ctrl+space to give the function name, and then add the `()` at the end.

The weight conversion program

Now, download the kilos to pounds program weights.py from the unit web-site (see the beginning of this document for the address). Right-click on the weights.py link and save the file to your Intprog folder.) Open the file in the editor (using with the File menu or the file explorer). Execute this program using the run button or ctrl+F9. As before, the program is executed in the shell; this time however, the program includes code that asks you to supply some input. Enter a number, and the program will give you the output before ending.

Let's make the weights program a little bit more complete by adding a few ***comment lines*** at the top of the program:

```
#-------------------------------------------------------
# A kilograms to pounds conversion program
# your name
# your six-digit student number
# Autumn Teaching Block 2013
#-------------------------------------------------------
```

Python will ignore any text that appears to the right of a # symbol, so this is how we ***document*** a program to make it more readable to yourself and fellow programmers. Execute

the program again to see that the comment lines haven't changed what the program does.

Looking ahead: lists and loops

To make things a little more interesting, let's take a sneak preview of some of the programming concepts that we'll study in depth later in the unit. First, using the shell, enter the following expressions at the shell prompt:

```
[3,6,1,2]
list(range(5))
list(range(10))
```

Each of these expressions is a ***list***: a kind of sequence of values. The first list has four numbers in it; the others are calculated using the function `range` that generates a given number of values starting from 0.

Now try:

```
for i in [3,6,1,2]:
    print(i)

for i in range(5):
    print(i)

for i in range(10):
    print(i)
```

You'll need to press return twice after each print statement to tell the shell that you've finished. These statements are known as ***loops***. Loops are used in programming to execute a piece of code over and over again. Try to guess what is happening when this code is being executed. The `i` is just a variable that is created and used by the loop. This variable is given each of the values in the sequences in turn and the print statement executed. Experiment with loops (using various lists and ranges) until you can predict their behaviour.

Program files with many functions

Now, download the pract01.py program from the unit web-site and save it into your Intprog folder. Open this program in the editor window. As you can see, this program contains three function definitions: `sayHello` (as seen already), `count` and `kilos2pounds`. The `count` and `kilos2pounds` functions contain code you've seen before. The blank lines between the function definitions are there just to make the code easier to read.

Execute this program in the normal way. Note that the program gives no output. Instead, it has defined three functions which can now be used (called) from the shell. Try typing the following at the >>> prompt:

```
sayHello()
count()
kilos2pounds()
```

This is how you'll write and execute your code for the next few practical worksheets. For each worksheet you will construct a single program file containing several function definitions, and you'll test this code by calling the functions from the shell.

Configure PyScripter's editor

PyScripter has many settings that you can change. For now, let's change the "template code" that appears whenever you create a new Python program. From the Tools menu, select Options and then File Templates. Click on Python Script, and then modify the text in the Template area so that it looks like:

```
#--------------------------------------------------------
#
# your name
# your six-digit student number
# Autumn Teaching Block 2013
#--------------------------------------------------------
```

with nothing except a single blank line below. Now click on Update, and then click OK to save this setting. To check this works, create a new Python file (e.g. from the File menu, choose New then New Python module). Your template should appear ready for you to add a description of the file on the top line, and your code below these comments.

Programming exercises

Each practical worksheet will include a set of programming exercises, and your attempts at the worksheet exercises will provide a basis for some of the unit assessment in later weeks. It is important therefore that you keep up-to-date with the practical exercises. As already mentioned, work at your own pace — you are not expected to complete the worksheet during a single practical session. You should, however, attempt to complete as much as you can in your own time before the following week's practical.

Each practical worksheet will begin with comparatively simple exercises, and will get more difficult towards the end. (Don't worry if you cannot do the questions marked [harder]. These are mainly intended for those students who would like a further challenge.) Remember that learning to program is fundamentally about developing your problem solving skills — the exercises are thus designed to make you think, and may take some time to complete.

After each exercise, remember to re-execute the program from the editor before testing the corresponding function in the shell. Begin by modifying the comments at the top of the pract01.py file to include your name and student (six digit) number. Add your functions to this file.

1. Write a function called `sayName` that displays your name.

2. Write a `sayHello2` function that uses two print statements to display the text:

   ```
   Hello
   World
   ```

3. Write a function `euros2pounds` which converts an amount in Euros entered by the user to a corresponding amount in Pounds. Assume that the exchange rate is 1.15 Euros to the Pound. (Hint: be sure to test your solution carefully.)

4. Write an `addUp` function that asks the user to enter two numbers, and outputs their sum.

5. Write a `changeCounter` function. This should ask the user how many 1p, 2p and 5p coins they have (using three separate questions), and then display the total amount

of money in pence. (Hint: remember that variable names cannot begin with a digit, so you might like to use names like `twoPence`.)

6. Write a `tenHellos` function that uses a loop to display "hello, world!" ten times (on separate lines).

7. Change the `count` function so that instead of counting from 0 to 9, it counts up from 1 to 10. (Hint: Use a little arithmetic in the print statement.)

8. [harder] Write a function `weightsTable` that outputs a two-column table of kilogram weights and their pound equivalents for kilogram values 0, 10, 20 ... 100. (Don't worry too much about formatting the table neatly.)

9. [harder] Write a `futureValue` function that uses a loop to calculate the future value of an investment amount, assuming an annual interest rate of 5.5%. The function should ask the user for the initial amount and the number of years that it is to be invested, and should output the final value of the investment using compound interest with the interest compounded every year.