TMTplus Introduction to Scientific Programming

Mahdi Farnaghi, Mahdi Khodadadzadeh & Robert Ohuru

March 2021

Note from the teaching staff

The materials that we use for this course are somewhat new and have been heavily reworked. We will likely have made mistakes in this work or have glossed over issues that deserved better or different treatment. Where you find issues worth noting please do report these to us as it will allow to repair and improve.

By the way, welcome to the Exercise book for Introduction to Scientific Programming! We believe this book is self-explanatory, so go ahead and practice.

Exercise 2

Functions

A fundamental skill in coding/scripting is the ability to tackle a complex problem by dividing it up into smaller problems that can be tackled one by one. In that context, function definitions and invocations are key.

The exercises

The main goal of each exercise is to use the Python programming language to review concepts you have learned during the lectures. In particular, you will work with *functions* in this exercise.

Also, from now on, we will use PyCharm IDE in our exercises. Make sure you type the code and do not copy and paste it. Copy and paste can generate bad characters, especially the “ and ” characters.

2.1 A wonderful world!

Ex 2.1 What does the following produce in the Python interpreter?

```
>>> len('How␣many␣characters␣do␣I␣have?')
```

The `str()` function converts an object into a string, therefore if we have an integer number, we can convert it to string using `str(number)`. Put the following statements in a script called `wonderful_1.py` and execute it. As you type in the code, try understand what it aims to do. Also, you may be making typos; essentially this will happen at some point eventually. Some typos are pretty

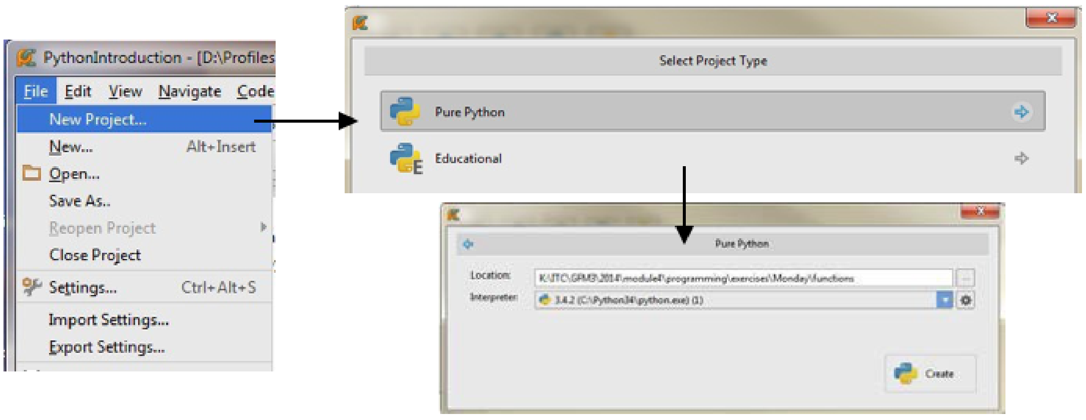
Ex 2.2

harmless, and others can break your code and cause an error message. Do not panic, read the message carefully, and act accordingly.

```
x = 'I␣see␣skies␣of␣blue␣and␣clouds␣of␣white'
n=str(len(x))
print(n+"␣-␣"+x)

x = 'The␣bright␣blessed␣day,␣the␣dark␣sacred␣night'
n=str(len(x))
print(n+"␣-␣"+x)

x = 'And␣I␣think␣to␣my␣self:'
n=str(len(x))
print(n+"␣-␣"+x)

x = '␣"What␣a␣wonderful␣world!"␣'
n=str(len(x))
print(n+"␣-␣"+x)
```

Ex 2.3

To make things easier you can use PyCharm. Open PyCharm and create a new Pure-Python project (under File). Name your project, and choose the Python interpreter. Next, add a new script: (using the File menu) and add a Python file called `wonderful_1` to your project.



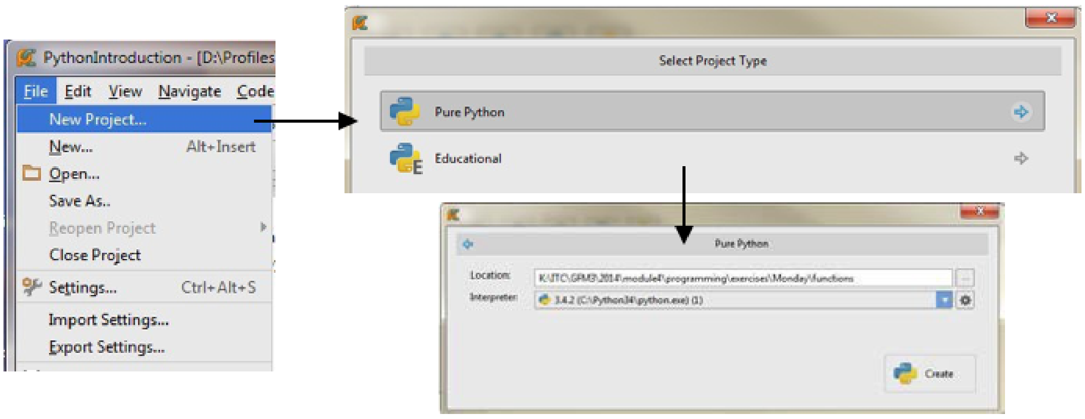
Location: K:\ITC\GFM3\2014\module4\programming\exercises\Monday\functions
Interpreter: 3.4.2 (C:\Python34\python.exe) (1)

Figure 2.1: The creation of a new PyCharm project

Important warning: The name of your `.py` file should never be equal to the name of any installed module, for example `math` or `numpy`. You can use names like `mathproject.py` or `numpyproject.py` but never `math.py`, `numpy.py` etc.

Write your code, save your file and run the code. Once you have, try find alternative ways to running code. There are at least two more.

If there are syntax errors, the code will not run and the Python interpreter will inform about the error. Read the error message, find the error and fix the code; next, re-run it. If you do not understand the error message (something not



Location: K:\ITC\GFM3\2014\module4\programming\exercises\Monday\functions
Interpreter: 3.4.2 (C:\Python34\python.exe) (1)

Figure 2.2: The program in a code pane

to be ashamed of as error messages can be notoriously cryptic sometimes), then use a search engine with the error message text and the extra term 'Python' and look for an explanation. In any case, pause for a moment and internalize the error message text. It is likely you will see it again in the future . . .

What do you observe when you run the code? Explain how the script works.

Ex 2.4 Incidentally, above we mentioned *syntax errors*. Try provide a definition of that term. There is a second category, named *semantic errors*; how would one define this term?

Observe that the print statement does not change and is in fact repeated four times. Wouldn't it be nice to be able to simplify the script a little bit? Functions can come to the rescue here.

2.2 My first function

Ex 2.5 Modify your script, and create `wonderful_2.py` by making it look like this.

```
def wonder_print(s):
    n = str(len(s))
    print(n + "␣-␣" + s)

wonder_print('I␣see␣skies␣of␣blue␣and␣clouds␣of␣white')
wonder_print('The␣bright␣blessed␣day,␣the␣dark␣sacred␣night')
wonder_print('And␣I␣think␣to␣my␣self:')
wonder_print('␣"What␣a␣wonderful␣world!"␣')
```

That is a lot better! We used what is called a function definition, subsequently used the function, and now the script looks cleaner, shorter (with fewer lines of code) and the code is also easier to read. Run it and observe the output.

Taking an existing program and trying to simplify the code by re-using similar pieces of code is called *refactoring*.

Why is refactoring good? Because it reduces repetition and improves readability of your program. Moreover, as a consequence there is less chance of making coding mistakes. This is why functions are useful.

2.3 More functions

Ex 2.6

Add a new Python file called brackets.py to your project and in it, define a function as follows:

```
def print_with_brackets(name):
    print ("[[[␣", name, "␣]]]")
```

Check how the function works and run it. Test it by printing your name.

Ex 2.7

Now try to write a function called print_three_names(), which takes three arguments and displays the following behaviour: Add to your file brackets.py and define a function that will have the following behaviour:

```
>>> print_three_names('Peter', 'Paul', 'Mary')
[[[ Peter ]]]
[[[ Paul ]]]
[[[ Mary ]]]
```

2.4 Changing arguments

Ex 2.8

What happens if you do the following statements?

1. print_with_brackets('GFM')
2. print('GFM' + 10)
3. print('GFM' , 10)
4. print_with_brackets('GFM') + 10
5. print('GFM' * 10)

6. print_with_brackets('GFM' * 10)

7. print_with_brackets('GFM') * 10

What are your conclusions? Is there any difference between + and * operators? Why does (2) give an error and does (3) work fine?

2.5 Going round in circles

In the below exercise, the code is already provided; but unfortunately the lines are mixed up. So, the challenge is e to put the lines in the correct order.

The aim is to define a function called `area_circle` that calculates the area of a circle with a given radius. Also define the function `squared`. Check `area_circle` by calling it with a value 5 for the radius. You will need to think about proper indentation as well.

```
return x**2
def squared(x):
def area\_circle(radius):
print (area\_circle(5))
import math
return squared(radius)*math.pi
```

2.6 Documenting things

It is high time that we look at documenting what we did. In fact, we are already too late. Good practice is to *document while you code*. The most important reasons for this are

1. documenting requires you to put into words the rationale of your code and this forces you to be precise in both the written word and the written code, that is the documentation process forces you to think,

2. at the time of coding, you are deepest involved in the code purpose, approach and semantics, so this is appropriate timing to also write the explanation of the code, and

3. providing documentation afterwards may lead to not documenting at all, or to insufficient and incomplete documentation.

x 2.9 Read an introduction to 'documentation strings', better known as *docstrings* here first: intro Python docstrings. This is 9 lines of text. Then next, read all about docstrings in PEP-257 and save that url for later references as a bookmark in your browser. PEP, by the way, stands for a Python Enhancement Proposal.

Now to finish off, go back to function `area_circle()` and provide a docstring for it in that file/code.