Programming Fundamentals

Testing, main, global variables

João Correia Lopes

INESC TEC, FEUP

23 October 2018

GOALS

By the end of this class, the student should be able to:

- Understand and perform (simple) unit testing
- Understand the use of the function `main()`
- Understand the use of global variables

BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3, 2018 (Chapter 4) [PDF]
- Brad Miller and David Ranum, Learning with Python: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (Section 6.7) [HTML]
- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3 (RLE), 2012 (Section 6.7) [HTML]

TIPS

- There's no slides: we use a script and some illustrations in the class. That is NOT a replacement for **reading the bibliography** listed in the *class plan*
- "Students are responsible for anything that transpires during a class—therefore **if you're not in a class**, you should get notes from someone else (not the instructor)"—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the class)
- We will be using **Moodle** as the primary means of communication

Contents

1 Unit testing

2 Using a Main Function

3 Tricks & Tips

4 Global variables

5 Computational Thinking

6 Exercises

6.7. UNIT TESTING

- It is a common best practice in software development to include automatic `unit testing` of source code
- Unit testing provides a way to automatically verify that individual pieces of code, such as functions, are working properly
- This makes it possible to change the implementation of a function at a later time and quickly test that it still does what it was intended to do
- Unit testing also forces the programmer to think about the different cases that the function needs to handle
- Extra code in your program which is there because it makes debugging or testing easier is called **scaffolding**
- A collection of tests for some code is called its *test suite*

UNIT TESTS

- At this stage we're going to ignore what the Python community usually does (see Codeboard), and we're going to code two simple functions ourselves
- Then we'll use these for writing our *unit tests*
- and we'll apply a *test suit* to `absolute_value()`
- First we plan our tests (think carefully about the "edge" cases):
  1. argument is negative
  2. argument is positive
  3. argument is zero

HELPER FUNCTION

We're going to write a helper function for checking the results of one test.

```
1  import sys
2
3  def test(did_pass):
4      """  Print the result of a test.  """
5      linenum = sys._getframe(1).f_lineno  # the caller's line number
6      if did_pass:
7          msg = "Test at line {0} ok.".format(linenum)
8      else:
9          msg = "Test at line {0} FAILED.".format(linenum)
10     print(msg)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/09/tests.py

TEST SUIT

With the helper function written, we can proceed to construct our *test suite*:

```
1  def test_suite():
2      """
3      Run the suite of tests for code in this module (this file).
4      """
5      test(absolute_value(17) == 17)
6      test(absolute_value(-17) == 17)
7      test(absolute_value(0) == 0)
8      test(absolute_value(3.14) == 3.14)
9      test(absolute_value(-3.14) == 3.14)
10
11 test_suite()         # Here is the call to run the tests
```

See it with the `absolute_value(n)` buggy version.

⇒ https://github.com/fpro-admin/lectures/blob/master/09/tests.py

USING A MAIN FUNCTION

- Using functions is a good idea
- It helps us to modularize our code by breaking a program into logical parts where each part is responsible for a specific task
- For example, remember the function called `drawSquare()` that was responsible for having some turtle draw a square of some size
- The actual turtle and the actual size of the square were defined to be provided as parameters
- These final five statements of the program perform the main processing that the program will do

⇒ https://github.com/fpro-admin/lectures/blob/master/09/mainproc.py

PROGRAM STRUCTURE

- Now our program structure is as follows
  1. First, import any modules that will be required
  2. Second, define any functions that will be needed
  3. Third, define a main function that will get the process started
  4. And finally, invoke the main function (which will in turn call the other functions as needed)
- In Python there is nothing special about the name `main`[1]

[1] We could have called this function anything we wanted. We chose main just to be consistent with some of the other languages.

ADVANCED TOPIC

- Before the Python interpreter executes your program, it defines a few special variables
- One of those variables is called `__name__`
- and it is automatically set to the string value `"__main__"` when the program is being executed by itself in a standalone fashion
- On the other hand, if the program is being imported by another program, then the `"__main__"` variable is set to the name of that module
- This ability to conditionally execute our main function can be extremely useful when we are writing code that will potentially be used by others
- For example, if we've collection of functions to do some simple math...

⇒ https://github.com/fpro-admin/lectures/blob/master/09/math.py

TIP: NONE IS NOT A STRING

- Values like `None`, `True` and `False` are not strings: they are special values in Python
- Keywords are special in the language: they are part of the syntax
- So we cannot create our own variable or function with a name `True` — we'll get a syntax error
- Built-in functions are not privileged like keywords: we can define our own variable or function called `len`, but we'd be silly to do so!

TIP: UNDERSTAND WHAT THE FUNCTION NEEDS TO RETURN

- Perhaps functions return nothing
  - some functions exists purely to perform actions, rather than to calculate and return a result (*procedures*)
- But if the function should return a value
  - make sure all execution paths do return the value

TIP: USE PARAMETERS TO GENERALIZE FUNCTIONS

- Understand which parts of the function will be hard-coded and unchangeable, and
- which parts should become parameters so that they can be customized by the caller of the function

TIP: TRY TO RELATE PYTHON FUNCTIONS TO IDEAS WE ALREADY KNOW

- In math, we're familiar with functions like `f(x) = 3x + 5`
- We already understand that when we call the function `f(3)` we make some association between the parameter `x` and the argument `3`
- Try to draw parallels to argument passing in Python

TIP: THINK ABOUT THE RETURN CONDITIONS OF THE FUNCTION

- Do I need to look at all elements in all cases?
- Can I shortcut and take an early exit?
- Under what conditions?
- When will I have to examine all the items in the list?

⇒ https://github.com/fpro-admin/lectures/blob/master/09/manyodd.py

TIP: GENERALIZE YOUR USE OF BOOLEANS

- Mature programmers won't write `if is_prime(n) == True:`
  - when they could say instead `if is_prime(n):`
- Like arithmetic expressions, booleans have their own set of operators (`and`, `or`, `not`) and values (`True`, `False`) and can be assigned to variables, put into lists, etc.

LOCAL VARIABLES

- Tip: Local variables do not survive when you exit the function
- Tip: Assignment in a function creates a local variable

As soon as the function returns (whether from an explicit return statement or because Python reached the last statement), the stack frame and its local variables are all destroyed.

GLOBAL VARIABLES

- UNLESS it makes use of variables that are **global**[2]

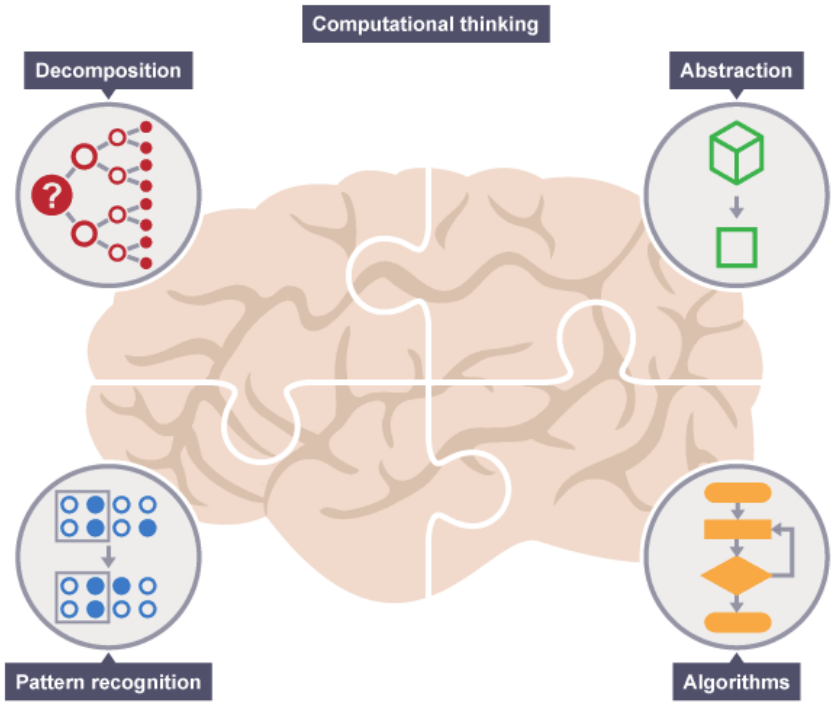
```
1  sz = 2
2  def h2():
3      """ Draw the next step of a spiral on each call. """
4      global sz
5      tess.turn(42)
6      tess.forward(sz)
7      sz += 1
```

Each time we call `h2()` it turns, draws, and increases the global variable `sz`

[2] It's generally considered bad practice to use global variables. Functions should be as self-contained as possible (**no side-effects**).

COMPUTATIONAL THINKING

Computational thinking allows us to take a complex problem, understand what the problem is and develop possible solutions. We can then present these solutions in a way that a computer, a human, or both, can understand.



⇒ BBC, Bitsize, Introduction to computational thinking

USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

DECOMPOSITION — breaking down a complex problem or system into smaller, more manageable parts

PATTERN RECOGNITION — looking for similarities among and within problems[3]

ABSTRACTION — focusing on the important information only, ignoring irrelevant details[4]

ALGORITHMS — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

⇒ BBC, Bitsize, Introduction to computational thinking

[3] Have any of the issues we've encountered in the past had solutions that could apply here?

[4] To make solutions as general as possible.

USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

DECOMPOSITION — breaking down a complex problem or system into smaller, more manageable parts

PATTERN RECOGNITION — looking for similarities among and within problems[3]

ABSTRACTION — focusing on the important information only, ignoring irrelevant details[4]

ALGORITHMS — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

⇒ BBC, Bitsize, Introduction to computational thinking

[3] Have any of the issues we've encountered in the past had solutions that could apply here?

[4] To make solutions as general as possible.

USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

DECOMPOSITION — breaking down a complex problem or system into smaller, more manageable parts

PATTERN RECOGNITION — looking for similarities among and within problems[3]

ABSTRACTION — focusing on the important information only, ignoring irrelevant details[4]

ALGORITHMS — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

⇒ BBC, Bitsize, Introduction to computational thinking

[3] Have any of the issues we've encountered in the past had solutions that could apply here?

[4] To make solutions as general as possible.

USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

DECOMPOSITION — breaking down a complex problem or system into smaller, more manageable parts

PATTERN RECOGNITION — looking for similarities among and within problems[3]

ABSTRACTION — focusing on the important information only, ignoring irrelevant details[4]

ALGORITHMS — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

⇒ BBC, Bitsize, Introduction to computational thinking

[3] Have any of the issues we've encountered in the past had solutions that could apply here?

[4] To make solutions as general as possible.

USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

DECOMPOSITION — breaking down a complex problem or system into smaller, more manageable parts

PATTERN RECOGNITION — looking for similarities among and within problems[3]

ABSTRACTION — focusing on the important information only, ignoring irrelevant details[4]

ALGORITHMS — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

⇒ BBC, Bitsize, Introduction to computational thinking

[3] Have any of the issues we've encountered in the past had solutions that could apply here?

[4] To make solutions as general as possible.

EXERCISES

- Moodle activity at: LE09: Testing, main and globals