Programming Fundamentals

Fruitful functions

João Correia Lopes

INESC TEC, FEUP

18 October 2018

Goals

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

- Identify functions that return a value (fruitful functions)
- Enumerate the diverse uses of the `return` statement
- Describe and use boolean functions
- Describe and use incremental program development
- Identify uses of function composition
- Enumerate the main PEP8 rules for writing Python programs

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]
- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3 (RLE), 2012 (Chapter 6) [HTML]
- Brad Miller and David Ranum, Learning with Python: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (Chapter 6) [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

RETURN VALUES

- The built-in functions we have used, such as `abs`, `pow`, `int`, `max`, and `range`, have produced results
- Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression

```
1   biggest = max(3, 7, 2, 5)
```

```
1   x = abs(3 - 11) + 10
```

- We are going to write more functions that return values, which we will call *fruitful functions*, for want of a better name.

RETURN VALUES

- The built-in functions we have used, such as `abs`, `pow`, `int`, `max`, and `range`, have produced results
- Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression

```
1  biggest = max(3, 7, 2, 5)
```

```
1  x = abs(3 - 11) + 10
```

- We are going to write more functions that return values, which we will call *fruitful functions*, for want of a better name.

RETURN VALUES

- The built-in functions we have used, such as `abs`, `pow`, `int`, `max`, and `range`, have produced results
- Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression

```
1  biggest = max(3, 7, 2, 5)
```

```
1  x = abs(3 - 11) + 10
```

- We are going to write more functions that return values, which we will call *fruitful functions*, for want of a better name.

THE RETURN STATEMENT

- In a fruitful function the return statement includes a **return value**
- This statement means: evaluate the return expression, and then return it immediately as the result (the fruit) of this function
- Code that appears after a `return` statement[1] is called **dead code**

```
1  def area(radius):
2      """returns the area of a circle with the given radius."""
3      fruit = 3.14159 * radius ** 2
4      return fruit
```

⇒ https://github.com/fpro-admin/lectures/blob/master/08/returns.py

[1] or any other place the flow of execution can never reach

MORE RETURNS

- All Python functions return `None` whenever they do not return another value
- It is also possible to use a `return` statement in the middle of a `for` loop, in which case control immediately returns from the function

⇒ https://github.com/fpro-admin/lectures/blob/master/08/moreReturns.py
⇒ See it on pythontutor

Incremental Development

- To deal with increasingly complex programs, we are going to suggest a technique called incremental development
- The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time

- Suppose we want to find the *distance between two points*, given by the coordinates (x_1, y_1) and (x_2, y_2)
- By the Pythagorean theorem, the distance is:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

⇒ https://github.com/fpro-admin/lectures/blob/master/08/distance.py

Incremental Development

- To deal with increasingly complex programs, we are going to suggest a technique called incremental development
- The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time

- Suppose we want to find the *distance between two points*, given by the coordinates (x_1, y_1) and (x_2, y_2)
- By the Pythagorean theorem, the distance is:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

⇒ https://github.com/fpro-admin/lectures/blob/master/08/distance.py

Incremental development (2)

The key aspects of the process are:

1. Start with a working skeleton program and make small incremental changes
2. Use temporary variables to refer to intermediate values so that you can easily inspect and check them
3. Once the program is working, relax, sit back, and play around with your options

Goal:

A good guideline is to aim for making code as easy as possible for others to read

INCREMENTAL DEVELOPMENT (2)

The key aspects of the process are:

1. Start with a working skeleton program and make small incremental changes
2. Use temporary variables to refer to intermediate values so that you can easily inspect and check them
3. Once the program is working, relax, sit back, and play around with your options

GOAL:

A good guideline is to aim for making code as easy as possible for others to read

DEBUGGING WITH PRINT

- A powerful technique for debugging, is to insert extra `print` functions in carefully selected places in your code
- Then, by inspecting the output of the program, you can check whether the algorithm is doing what you expect it to
- Be clear about the following, however:
  - You must have a clear solution to the problem, and must know what should happen before you can debug a program
  - Writing a program doesn't solve the problem — it simply *automates* the manual steps you would take
  - avoid calling `print` and `input` functions inside fruitful functions, *unless the primary purpose of your function is to perform input and output*[2]

[2] The exception is the `print` statements for debugging, later removed

DEBUGGING WITH PRINT

- A powerful technique for debugging, is to insert extra `print` functions in carefully selected places in your code
- Then, by inspecting the output of the program, you can check whether the algorithm is doing what you expect it to
- Be clear about the following, however:
  - You must have a clear solution to the problem, and must know what should happen before you can debug a program
  - Writing a program doesn't solve the problem — it simply *automates* the manual steps you would take
  - avoid calling `print` and `input` functions inside fruitful functions, *unless the primary purpose of your function is to perform input and output*[2]

[2] The exception is the `print` statements for debugging, later removed

COMPOSITION

- **Composition** is the ability to call one function from within another
- As an example, we'll write a function that takes two points, the center of the circle (xc, yc) and a point on the perimeter (xp, yp), and computes the area of the circle

⇒ https://github.com/fpro-admin/lectures/blob/master/08/area.py

BOOLEAN FUNCTIONS

- **Boolean functions** are functions that return Boolean values
  - which is often convenient for hiding complicated tests inside functions[3]

```
1  def is_divisible(x, y):
2      """ Test if x is exactly divisible by y """
3      if x % y == 0:
4          return True
5      else:
6          return False
```

```
1  if is_divisible(x, y):
2      ... # Do something ...
3  else:
4      ... # Do something else ..
```

[3] Can we avoid the `if`?

BOOLEAN FUNCTIONS

- **Boolean functions** are functions that return Boolean values
  - which is often convenient for hiding complicated tests inside functions[3]

```
1  def is_divisible(x, y):
2      """ Test if x is exactly divisible by y """
3      if x % y == 0:
4          return True
5      else:
6          return False
```

```
1  if is_divisible(x, y):
2      ... # Do something ...
3  else:
4      ... # Do something else ..
```

[3] Can we avoid the `if`?

PEP 8 — Style Guide for Python Code

- use 4 spaces (instead of tabs) for indentation
- limit line length to 78 characters
- when naming identifiers use `lowercase_with_underscores` for functions and variables
- place *imports* at the top of the file
- keep function definitions together below the `import` statements
- use *docstrings* to document functions
- use two blank lines to separate function definitions from each other
- keep top level statements, including function calls, together at the bottom of the program
- tip: Spyder3 may help you complying with PEP8...

EXERCISES

- Moodle activity at: LE08: Fruitful functions