Programming Fundamentals

Modules, Help, Trace & tips

João Correia Lopes

INESC TEC, FEUP

11 October 2018

Goals

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

- Describe an overview of the Modules available in the Python Standard Library
- Describe the contents of the `math` and `random` modules
- Use the Python Help and understand its meta-notation
- Debug Python programs
- Trace a program

BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3, 2018 (sections: 3.3.6, 3.3.8, 3.4) [PDF]
- Brad Miller and David Ranum, Learning with Python: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (chapters: 5, 3) [HTML] [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 sheet*
- "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

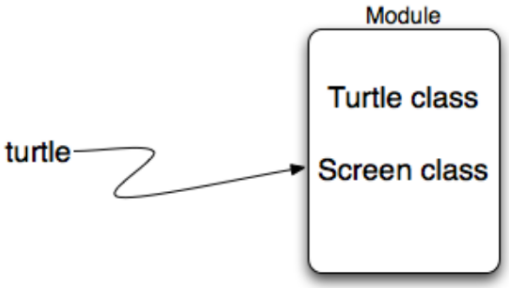
PYTHON MODULES

- A **module** is a file containing Python definitions and statements intended for use in other Python programs
- There are many Python modules that come with Python as part of the standard library
- We have already used one of these quite extensively, the turtle module
- Recall that once we import the module, we can use things that are defined inside

⇒ https://docs.python.org/3.6/
⇒ https://docs.python.org/3.6/library/
⇒ https://docs.python.org/3/py-modindex.html
⇒ https://docs.python.org/3.6/library/turtle.html
⇒ https://github.com/python/cpython/blob/3.6/Lib/turtle.py

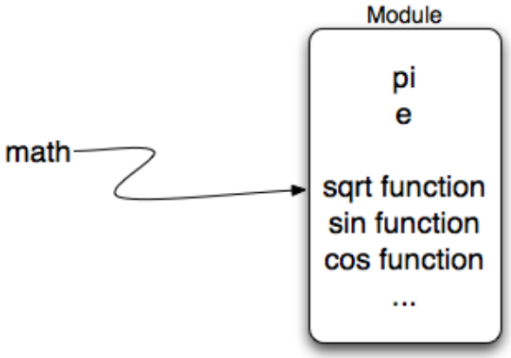
USING MODULES

- The first thing we need to do when we wish to use a module is perform an import
- The statement `import turtle` creates a new name, `turtle`, and makes it refer to a module object
- This looks very much like the reference diagrams for simple variables



THE MATH MODULE

- The math module contains the kinds of mathematical functions you would typically find on your calculator, and
- some mathematical constants like *pi* and *e*



⇒ https://github.com/fpro-admin/lectures/blob/master/06/math.py

THE RANDOM MODULE

We often want to use **random numbers** in programs[1].
Here are a few typical uses:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin
- To shuffle a deck of playing cards randomly
- To randomly allow a new enemy spaceship to appear and shoot at you
- To simulate possible rainfall when we make a computerised model for estimating the environmental impact of building a dam
- For encrypting your banking session on the Internet

⇒ https://github.com/fpro-admin/lectures/blob/master/06/random.py

[1] It is important to note that random number generators are based on a *deterministic algorithm* — repeatable and predictable. So they're called *pseudo-random generators* — they are not genuinely random.

3.3.8 HELP AND META-NOTATION

- Python comes with extensive documentation for all its built-in functions, and its libraries.
- See for example docs.python.org/3/library/...range
- The square brackets (in the description of the arguments) are examples of *meta-notation* — notation that describes Python syntax, but is not part of it
  - **range**([*start*,] *stop* [, *step*])
  - **for** *variable* **in** *list* :
  - **print**([*object*, ...])
- Meta-notation gives us a concise and powerful way to describe the *pattern* of some syntax or feature.

How to be a Successful Programmer

- One of the most important skills you need to aquire is the ability to debug your programs
- Debugging is a skill that you need to master over time
- As programmers we spend 99% of our time trying to get our program to work
- But here is the secret, when you are successful, you are happy, your brain releases a bit of chemical that makes you feel good
- **Start small, get something small working, and then add to it**

How to Avoid Debugging

Mantra

Get something working and keep it working

- **Start Small**
  - This is probably the single biggest piece of advice for programmers at every level.
- **Keep it working**
  - Once you have a small part of your program working the next step is to figure out something small to add to it.

Ok, let's look at an example: the `alarm_clock.py` of RE02

⇒ https://github.com/fpro-admin/lectures/blob/master/06/alarm_clock.py

Beginning tips for Debugging

Debugging a program is a different way of thinking than writing a program.
The process of debugging is much more like being a detective.

1. Everyone is a suspect (Except Python)!
2. Find clues
   - Error messages
   - Print statements

Beginning tips for Debugging

Debugging a program is a different way of thinking than writing a program.
The process of debugging is much more like being a detective.

1. Everyone is a suspect (Except Python)!
2. Find clues
   - Error messages
   - Print statements

SUMMARY ON DEBUGGING

- Make sure you take the time to understand error messages
  - They can help you a lot
- `print` statements are your friends
  - Use them to help you uncover what is **really** happening in your code
- Work backward from the error
  - Many times an error message is caused by something that has happened before it in the program
  - Always remember that python evaluates a program top to bottom

Tracing a program

- Tracing involves becoming the computer and following the *flow of execution* through a sample program run, recording the state of all *variables* and any *output* the program generates after each instruction is executed
- Let's try with the Collatz sequence of `3`

`3`	`3,`
`10`	`3, 10,`
`5`	`3, 10, 5,`
`16`	`3, 10, 5, 16,`
`8`	`3, 10, 5, 16, 8,`
`4`	`3, 10, 5, 16, 8, 4,`
`2`	`3, 10, 5, 16, 8, 4, 2,`
`1`	`3, 10, 5, 16, 8, 4, 2, 1.`

Problems with logic and flow of control

We often want to know if some condition holds for any item in a list, e.g. “does the list have any odd numbers?”

This is a common mistake:

```
numbers = [10, 5, 24, 8, 6]
# Buggy version
for number in numbers:
    if number % 2 == 1:
        print(True)
        break
    else:
        print(False)
        break
```

⇒ https://github.com/fpro-admin/lectures/blob/master/06/odds.py

Tip: Think about the return conditions of the loop

- 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/06/odds.py

Tip: Generalize your use of Booleans

- Programmers won't write `if is_prime(n) == True:` when they could say instead `if is_prime(n):`
- Think more generally about Boolean values, not just in the context of `if` or `while` statements.
- Like arithmetic expressions, they have their own set of operators (`and`, `or`, `not`) and values (`True`, `False`) and can be assigned to variables, put into lists, etc
- See: wikibooks

Tip: Don't create unnecessary lists

- Lists are useful if you need to keep data for later computation
- But if you don't need lists, it is probably better not to generate them
- In the example below, there's two versions and both work
- What reasons are there for preferring the second version here?

⇒ https://github.com/fpro-admin/lectures/blob/master/06/loops.py

EXERCISES

- Moodle activity at: LE06: Modules, Help, tips & tricks