CPSC 301: Computing in the Life Sciences
Lecture Notes 7:
Python Modules & Testing

2017-02-09

Jessica Dawson
jqdawson@cs.ubc.ca
http://www.cs.ubc.ca/~jqdawson

University of British Columbia
Department of Computer Science

2016 W2
Practice materials posted
  – Midterm practice questions (not exhaustive coverage)
  – Recommended exercises from textbook
    • Basically everything except topics we haven’t covered yet (e.g. Lists)

Lab 03 marks posted to Handback on Connect
  – (note that it’s out of 19 total)
• Reminder to take advantage of office hours for help with Lab05 after lab!
  – Several today and tomorrow
  – Schedule is always available on Piazza (pinned near top)

• Reminder no labs next week
  – Monday is a holiday (department closed)
  – 5-7pm Thursday will be an open lab
    • TAs will be there and anyone can attend to ask questions, get extra help or feedback on midterm preparation, etc.
Midterm Preparation

• Coverage:
  – L01 – Intro to L08 – Methods (and corresponding textbook readings)
  – Labs 01 – 05 (but note that you won’t be using computers on the midterm, so some lab learning goals won’t be relevant)

• To best prepare, you should:
  – Review the learning goals for each lecture and lab (as you go)
  – Do the practice exercises from the textbook
  – Then review / redo pre-class quizzes and in-class exercises and clickers (make sure you understand the answers)
  – Then do the midterm practice questions
Objectives

At the end of this section, you will be able to:

• Use `import` and related commands to access variables, functions and objects (and later methods and classes) from other files

• Write your own modules (actually, you have been doing this all along)

• Use the `doctest` module to automatically run the test examples in docstrings

• Design your own modules which will automatically test themselves when they are run but not when they are imported
Python Modules

• A module is a collection of (usually related) functions and data (variables) grouped in a single file with .py extension. The file name, (without the ".py"), is the name of the module.

• Python comes with hundreds of modules
  – math module has many functions like
    • sqrt (square root), trunc (trancate a float to int), sin
    • etc.

• To use the functions of a module we have to import it with the statement
  import module-name, or
  import module-name as local-name

• After importing a module, Python creates a variable with the module name (original or local, if defined) which references that module.

• To call a function in that module you have to use module-name.function-name as the function name.

• Example:
  import math
  var = 2.95
  math.tranc(var)
Python Modules (cont)

• A module may also have a set of data
  – i.e. the math module has
    \[ e = 2.7182818284590451 \]
    \[ pi = 3.1415926535897931 \]
    these are constant values that you can use, but must not change:
    \[ radius = 4.0 \]
    \[ circle_area = math.pi \times radius \times 2 \]

• You can import only what you want from a module using:

  from math import floor, pi
  \[ circle_area = floor(pi \times radius \times 2) \]

• But if you later import another function called floor() from another
  module only the last one will be available.

• All built-in functions are in the module \_\_builtins\_\_. This module is
  automatically imported

• To check the content of a module you can use:
  
  help(module)    - long comments
  dir(module)     - only names of items
Defining Modules

• Every Python file (with extension .py) is a module
  – every time you create a program and save it, you create a module
  – you can import and use this module in another program

• Example:
  – population growth function is in file pgrowth.py
  – to use it, the program birthrate has to import it

• How does Python finds the files that needs to import?
  – Python first looks in the directories that contain its libraries, called default directories
  – then it looks in the current directory
  – in Spyder the current directory is the directory that contains the last program that was executed
  – for now, we can import any files from the default or current directories, but not from any other directory (see later for more)
Importing Modules

• When a module is imported, Python executes it. We say that the module is loaded (in memory).

• When a module is imported again in the same session, Python does not load it again. It knows that it is already in.

• In many cases, we want to execute different parts of the module when it is imported than when we run it as the main program.

• Python defines a variable `__name__` (two underscores before and after) within each module that is set automatically.
  • By convention, implicit Python entities (entities such as variables and functions which Python creates automatically) are given names that begin and end with double underscore.

• When a module is run as the main program (i.e. directly in the console or by clicking the run button), its `__name__` variable is set to `'__main__'`. When a module is imported, its `__name__` variable is set to its name.

• Therefore, if we want to run some code only when a module runs as the main program we include that code in the conditional:

  ```python
  if __name__ == '__main__':
  ```

  • Typical use: Tests for the module. We don't want to execute them when the module is imported.
A module `foo`

- In the subsequent clicker questions, assume that `foo.py` contains the following lines:

```python
def bar(x):
    print('name in bar:', __name__)
    return x * 2

def baz(x):
    return x / 2

print('name is: ', __name__)
if __name__ == '__main__':
    print('in main')
else:
    print('not in main')
print('final line')
```
As they develop a unit of code (like a function, a module), programmers test the unit by performing *unit testing* on the unit.

A unit testing consists of a set of *test cases* for that unit.

A *test case* consists of the *input values* and the *expected output*.

A popular style of programming is the *test-driven development* (TDD):

- write the test cases before you write the code for the unit.

When all units have been developed and tested, we test the whole program to verify that satisfies the user requirements.

The testing of the whole program is usually called *system testing* and there are many types of that:

- *alpha testing* : done by developers
- *beta testing* : done by some selected users
- *installation testing* : done at the final site
- etc
Black-box Testing

• The type of unit testing that we usually perform in a test driven development is called *black-box testing*.

• In a black-box testing, the test cases for a unit are usually derived from the unit's specification and include:
  – *typical cases* which use typical values for the inputs
  – *boundary cases* that use boundary values: values that are the smallest or largest allowable values for the unit's inputs
  – *simplest interesting cases* whose input values are close to boundary values

• Examples of boundary and interesting cases:
  – For numbers: zero, positive and negative typical values, largest and smallest allowed values, one value away from the smallest and largest value
  – For collections (like strings, ranges, lists, etc): empty collection, collection with one element, collections with duplicate values
  – For searching: no match found, one match, many matches.
Example of Test Cases

- Example: The test cases for a unit testing of the `population` function in the `pgrowth` module:
  - typical case:
    `population(1000, 20, 10, 10)` should return 2594
  - boundary case:
    `population(1000, 0, 0, any periods)` should return 1000
  - boundary case:
    `population(1000, 0, 10, 1)` should return 900
  - boundary case:
    `population(1000, 10, 0, 1)` should return 1100
  - boundary case:
    `population(1000, any, any, 0)` should return 1000
  - interesting case:
    `population(1000, 20, 10, 1)` should return 1100

- `pgrowth_test.py` contains these test cases
doctest : A simple Python Library for Testing

- Programmers use available libraries to help them write and maintain their tests
- **doctest** is a very simple module for test that we can write within the doc string, exactly like the examples we use in our recipe
- **doctest**
  - searches the doc string for lines that start with `>>>`
  - executes them and
  - compares the output with the next line which supposed to have the expected result
- Comparison is exact string match (including any space or newline characters)
- To activate the doctest when the module is executed as main program, we need to use:
  ```python
  if __name__ == '__main__':
      import doctest
      doctest.testmod()
  ```
- Check `pgrowth_doctest.py` for an example.
Other Python Testing Frameworks

• There are major limitations to the `doctest` module
  – It is difficult to create test cases which require setting up any significant amount of data
  – Test for success is based on matching strings
• Python also has `unittest` and `nose` modules
  – You can define more complex test cases
  – Success can be defined in terms of any Boolean expression
• The `unittest` module is part of the Python standard library
  – Any installation of Python will include this module
• The `nose` module builds on top of `unittest`
  – Claims to be easier to build, run and maintain test suites
  – Also supports extensive collection of “plug-ins”
Summary

• A file with the `.py` extension is called a “module”
• A module can contain many related variables / functions / objects
• Python provides several ways of executing the commands in a module: `import` and `from ... import`
• The commands in a module are executed only once, no matter how many times it is imported
• The names of the variables / functions / objects defined in the other file depend on how it is imported
• A file can determine whether it has been imported or run directly by examining the `__name__` variable
• The `doctest` module can be used to automatically run the example tests in a module’s docstrings
• Python provides several other frameworks for automating more complicated test processes