Scientific Computing 272

Section 3: Modules in Python

Last updated: 10 April 2019

UNIVERSITEIT STELLENBOSCH UNIVERSITY

Willem Bester

Section Outline

Software Quality

Importing Modules

Defining Modules

Objects and Methods

Testing

Modular Software

- That old adage: Rome wasn't built in a day
- Nor are most programs written by one person only
- Stupid programmers try to do everything by themselves
- Clever programmers stand on the shoulders of giants
- Bottom line: If another programmer solved a problem well, don't reinvent the wheel[1]
- Code re-use is one of the most important lessons of software engineering
- Although this course is not about software engineering, we must still know what good software engineering principles are

[1] This, of course, does not necessarily apply to course work.

Software Quality

Characteristic	Description
Correctness	The degree to which the software adheres to its specific requirements
Reliability	The frequency and criticality of software failure
Robustness	The degree to which errors are handled gracefully
Usability	The ease with which users learn and execute tasks with the software
Maintainability	The ease of making changes
Reusability	The ease with which components can be used by other software systems
Portability	The ease of using software across multiple platforms
Efficiency	The degree to which software fulfils its purpose without wasting resources

Modules

- A **module** is a collection of functions that are grouped together in a single file
- Functions in a module are typically related in some way
- For example, the `math` module contains mathematical functions such as `cos` (cosine) and `sqrt` (square root)
- We will write our own modules
- We also learn about existing modules
- Remember, we don't want to reinvent the wheel
- But sometimes we want to build a better mouse trap

Importing Modules

- When your refer to someone else's work in a scientific paper, you have to cite it
- When you want to use a function in a module, you have to **import** it

Example

```
>>> import math
```

- Importing lets Python know you want to use the module
- It also loads the relevant file as if you've type it in
- You can also ask for help

Help for Imported Modules

Example

```
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.
```

Importing Modules

Example

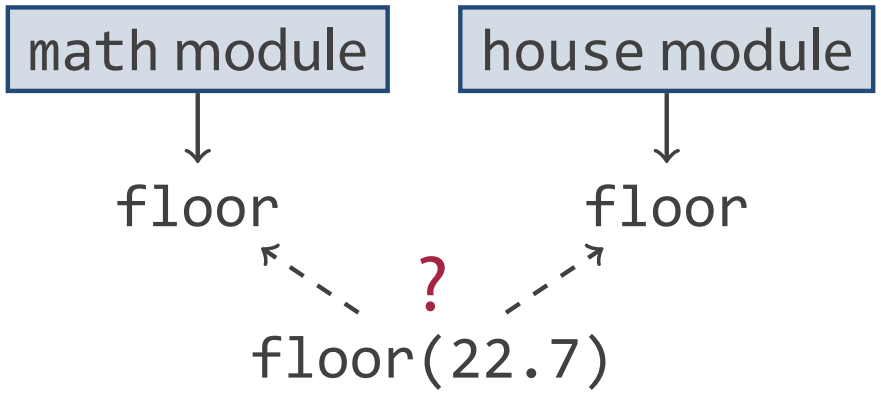
```
>>> sqrt(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

- Python doesn't know where to find `sqrt`
- **Qualify** the method name with the module name

Example

```
>>> math.sqrt(9)
3.0
```

Namespace Qualification



Qualifying the name is necessary since more than one module may contain a function with the same name.

Which function?

```
>>> import math
>>> import house
>>> floor(22.7)
```

Variables in Modules

- Once a module has been imported, it stays in memory until the program terminates
- Modules can contain more than functions
- `math`, for example, contains the variable `pi`

Example

```
>>> import math
>>> math.pi
3.141592653589793
>>> radius = 7
>>> print('area is {:.6f}'.format(math.pi * radius**2))
area is 153.938040
```

Variables and Constants

Example

```
>>> math.pi = 3
>>> radius = 7
>>> print('area is {:.6f}'.format(math.pi * radius**2))
area is 147.000000
```

- You can change the values of these variables
- But … DON'T!
- Many languages have **constants**
- The value of a constant cannot be changed after it has been defined
- That Python does not is a significant design flaw

Cherry-Picking Imports

- Using fully-qualified names is not always convenient
- You may specify exactly what you want to import

Example

```
>>> from math import sqrt, pi
>>> sqrt(8)
2.828427124746l903
>>> radius = 5
>>> print('circumference =', 2 * pi * radius)
circumference = 31.41592653589793
```

- Careful: Functions with the same name but from different modules may cause trouble
- The last to be imported replaces the previous ones

Cherry-Picking Imports

It is also possible to import everything from a module.

Example

```
>>> from math import *
>>> r = 7
>>> print('area = {:.6f}'.format(pi * r ** 2))
area = 153.938040
>>> sqrt(9)
3.0
```

- Quite often, however, this is not a good idea
- It is too easy for functions with the same name from different modules to clash

Defining Your Own Modules

- Define your own modules by putting functions into a file
- The name of the file must end with a `.py` extension
- Put the following in a file called `temperature.py`
- Remember to indent

Example (`temperature.py`)

```
def to_celsius(t):
    return (t - 32) * 5 / 9

def above_freezing(t):
    return t > 0
```

Congratulations! Your First Module!

Example

```
>>> import temperature
>>> temp = temperature.to_celsius(33.3)
>>> temperature.above_freezing(temp)
True
```

- Note: `t > 0` is a **boolean** expression
- Boolean values are of type `bool`, and may take either `True` or `False` as values
- We will consider the algebra of boolean expressions in a later section

What Happens during Import

- Experiment by putting the following in `experiment.py`

Example (`experiment.py`)

```
print("The panda's scientific name is 'Ailuropa melanoleuca'")
```

- Then import it

Example

```
>>> import experiment
The panda's scientific name is 'Ailuropa melanoleuca'
```

What Happens during Import

- Python executes modules as it imports them
- You can do anything in a module you can do in the Python interpreter
- Start a new Python session, and try the following:

Example

```
>>> import experiment
The panda's scientific name is 'Ailuropa melanoleuca'
>>> import experiment
>>>
```

- Note the message was not printed the second time

What Happens during Import

- Python only loads a module the first time it is imported
- Python keeps track of the modules it has already seen
- When Python encounters a module it has already imported, it simply skips over this module
- Doing so saves time
- Also, when we import modules that import other modules in turn, not importing a module more than once greatly increases performance
- While testing and debugging modules interactively, use the `importlib.reload(`⟨**module name**⟩`)` function if you have to "reimport" a module that has been updated

What Happens during Import

Example

```
$ python3
Python 3.7.2+ (default, Feb 14 2019, 22:48:45)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>> import experiment
The panda's scientific name is 'Ailuropa melanoleuca'
>>> import experiment
>>> from importlib import reload
>>> reload(experiment)
The panda's scientific name is 'Ailuropa melanoleuca'
<module 'experiment' from 'experiment.py'>
```

Using `__name__`

- Python files can either be run directly or be imported (and used) by another program
- It is sometimes useful to tell which module is the program invoked by a user
- Python defines a special variable called `__name__` in every module
- The double underscore is colloquially referred to as a "dunder" for "double underscore"
- It is usually used to indicate something special to Python
- It is a convention, and not part of the syntax

Example (`echo.py`)

```
print('echo: __name__ is', __name__)
```

Using __name__

- If we run `echo.py` directly

Example

```
$ python3 echo.py
echo: __name__ is __main__
```

- If we import `echo.py`:

Example

```
>>> import echo
echo: __name__ is echo
```

Using __name__

Example (import_echo.py)

```
import echo
print('After import, __name__ is', __name__)
print('And echo.__name__ is', echo.__name__)
```

Example

```
$ python3 import_echo.py
echo: __name__ is echo
After import, __name__ is __main__
And echo.__name__ is echo
```

Using `__name__`

- `__main__` means "this module is the main program"
- When Python import a module, it sets `__name__` of the module to the name of the module
- So, a module can tell whether it is the main program or not
- See what happens when you run the following directly and when you import it

Example (`test_main.py`)

```
if __name__ == '__main__':
    print('I am the main program')
else:
    print('Someone is importing me')
```

Providing Help

- Copy `temperature.py` to a new file `temp_round.py`
- Then modify `to_celsius` so that it rounds the result

Example (`temp_round.py`)

```
def to_celsius(t):
    return round((t - 32) * 5 / 9)

def above_freezing(t):
    return t > 0
```

Providing Help

Example

```
>>> import temp_round
>>> help(temp_round)
Help on module temp_round:

NAME
    temp_round

FILE
    /home/whkbester/wb272/temp_round.py

FUNCTIONS
    above_freezing(t)

    to_celsius(t)
```

This is not particularly helpful….

Docstrings

Let's add some **docstrings**, which is short for "documentation string".

Example

```
"""Functions for working with temperatures."""

def to_celsius(t):
    """Convert the temperature t from Fahrenheit
    to Celius."""
    return round((t - 32) * 5/ 9)

def above_freezing(t):
    """Return True if the temperature t in Celsius
    is above freezing; and False otherwise."""
    return t > 0
```

Docstrings

Example

```
>>> import temp_round
>>> help(temp_round)
Help on module temp_round:

NAME
    temp_round - Functions for working with temperatures.

FUNCTIONS
    above_freezing(t)
        Return True if the temperature t in Celsius
        is above freezing; and False otherwise.

    to_celsius(t)
        Convert the temperature t from Fahrenheit
        to Celius.

FILE
    /home/whkbester/wb272/temp_round.py
```

Objects and Methods

- We have already met **overloaded operators**—operators that change what they do based on their operand data types
- View it the other way around: Every data type has a set of operations defined on it
- For string, we have seen the concatenation (+) operator
- Single-character operators for more involved operations are impractical
- The solution: Objects and methods, which we study in detail later in the course

String methods

- A Python string "owns" a special set of functions, called **methods**, that define string operations: Every string automatically has all of the methods for the string data type
- Methods are called similarly to functions
- Something that has methods is called an **object**

Example (`capitalize()`)

```
>>> 'superman'.capitalize()
'Superman'
>>> villain = 'luthor'
>>> villain.capitalize()
'Luthor'
>>> villain
'luthor'
```

String methods

- Using methods is almost the same as using functions
- The difference is that a method almost always does something with its owner object

Example

```
>>> 'Lois'.startswith('l')
False
>>> 'Lois'.startswith('L')
True
>>> 'Lois'.endswith('s')
True
>>> 'Lois'.endswith('a')
False
```

Method Chaining

- We can **chain** multiple methods together by calling a method of the value returned by another method call
- For example, calling the method `swapcase` of some string returns a another string (that owns all of the string methods)

Example

```
>>> 'The Daily Planet'.swapcase()
'tHE dAILY pLANET'
>>> 'The Daily Planet'.swapcase().endswith('LANET')
True
```

How Method Chaining Works

- Python automatically creates a temporary variable to hold the value of the `swapcase()` method
- This value is a string (object) and, therefore, has the `endswith` method
- Once `endswith` returns, the string `'tHE dAILY pLANET'` returned by `swapcase`, is discarded—just as if we had typed an expression at the Python prompt without assigning it to a variable

Testing

- To ensure software quality—and the results of programmatic scientific analyses—programs should be tested
- **Quality assurance** (QA): checking that software is doing the right thing
- Put effort into QA $\implies$ more productive
- We use testing frameworks
  - Easy to test (and re-test when something has changed)
  - Easy for others to use

A Test Skeleton

Example (`test_temp_round.py`)

```
import nose
import temperature

def test_to_celsius():
    """Test the function to_celsius."""
    pass # fill in later

def test_above_freezing():
    """Test the function above_freezing."""
    pass # fill in later

if __name__ == '__main__':
    nose.runmodule()
```

The Nose Library

- Nose automatically looks for files with names that start with "`test_`"

Contents of a Nose test module

1. Statements to import Nose and the module to be tested
2. Functions that actually test the module
3. A function to trigger execution of these test functions

- The name of each test function must also start with "`test_`"
- Now, run the test module….

The Nose Library

Example

```
$ python3 test_temperature.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

- The `pass` statement is just a placeholder and does nothing
- The two dots mean that the two tests ran successfully
- If a test fails, Nose prints an `F`

An Example of a Unit Test File

Example (`test_to_celsius.py`)

```
import nose
from temp_round import to_celsius

def test_freezing():
    """Test freezing point."""
    assert to_celsius(32) == 0

def test_boiling():
    """Test boiling point."""
    assert to_celsius(212) == 100

def test_roundoff():
    """Test that roundoff works."""
    assert to_celsius(100) == 38 # NOT 37.77

if __name__ == '__main__':
    nose.runmodule()
```

Nose Test Outcomes

- We test by comparing the **actual value** returned by a function with the **expected value** (that it's supposed to return)
- We use the `assert` statement to state what we believe to be true—here, that the returned value must be equal to the actual value

Test outcomes

1. **Pass**: The actual value matches the expected value
2. **Fail**: The actual value is different from the expected value
3. **Error**: Something went wrong inside the test itself, that is, the test case contains a bug … in this case the test tells us nothing about the system being tested