Computational Thinking

Modules

In this class …

- What is a module?
- What are some of the Python standard modules?
- How can we make a module available to our program?
- How can we access the members of a module?
- How can we create our own modules?
- How can we prevent “loose” code to run upon importation of modules?

Not reinventing the wheel

- Say we need to calculate `sin(π)`
- The formula for the sine function is (approx.):

$$sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$$

- using the Maclaurin Series
- We could create a function to implement `sin`
- Because, it is a very common function, someone already did it for us and made it available through a **module**

Modules

- A module is a collection of variables and functions grouped together in a single file
- The variables and functions should be related
- `math` is a module that has variables and functions related to math (dah!)
- There, we can find functions like `sin`, `cos`, and variables like `pi` and `e`
- In order to access them, the module must be imported

Module *vs.* Package

- A package is, simply, a directory with modules
- Less simply, it requires the inclusion of a special file (not in the scope of this module)
- Popular packages:
  - `numpy`: scientific computing
  - `matplotlib`: plots
  - `tkinter`: graphical interfaces

Standard Python Library Back to the Modules

- The SPL has hundreds of modules. Examples are:
  - `cmath`: equivalent to `math`, but for complex numbers
  - `datetime`: working with dates and times
  - `sys` and `os`: operating system related
  - `urllib`: internet related
  - `random`: random number generation
  - `re`: regular expressions
  - `string`: useful string constants

Importing a module

```
>>> import math
```

- A variable named math of type module is created (check it: `type(math)`)

- To get help for the module (we already know this):

```
>>> help(math)
…
    sin(…)
          sin(x)

          Return the sine of x
 (measured in radians)
…
    DATA
…
          pi = 3.141592653589793
…
```

Using the module

```
>>> sin(pi)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in
<module>
    sin(pi)
NameError: name 'sin' is not defined
```

Using the module ...

```
NameError: name 'sin' is not defined
```

- What went wrong?
- `sin` is “inside” variable `math`, so we must provide the “path” to function `sin` explicitly
- The “path” is provided by the operator `.` (dot):

```
>>> math.sin(pi)
```

Second attempt

```
>>> math.sin(pi)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    math.sin(pi)
NameError: name 'pi' is not defined
```

- How can we fix this?

1 minute problem

- Use IDLE to calculate the value of the following expression:

$$\sqrt{4!\pi^2}$$

- Solution 1:

```
>>> (4*3*2*1 * 3.1415 *
 3.1415)**(1/2)
 15.390144053906708
```

- Solution 2:

```
>>> math.sqrt(math.factorial(4) *
 math.pi**2)
 15.390597961942369
```

- Solution 3:

```
>>> math.sqrt(math.factorial(4) *
pow(math.pi, 2))
15.390597961942369
```

Making expressions shorter

```
math.sqrt(math.factorial(4) * pow(math.pi, 2))
```

can be made shorter if we import just what we need

- To import what we need:

```
from math import sqrt, factorial,
 pi
```

- It looks better now:

```
sqrt(factorial(4) * pow(pi, 2))
```

Why don’t just import all?

- It is possible to import all the members of a module:

```
>>> from math import *
```

- What is the problem with that?
- Variables and functions with the same name will be replaced (even built-ins)
- If you want to import everything from the module, just import the module and use the module name as a prefix (e.g. `import math`)

The danger

```
>>> help(pow)
Help on built-in function pow in module
builtins:

pow(x, y, z=None, /)
    Equivalent to x**y (with two
arguments) or x**y % z
        (with three arguments)

    Some types, such as ints, are able to
use a more efficient
        algorithm when invoked using the
three argument form.
```

```
>>> from math import *
>>> help(pow)
Help on built-in function pow in module
math:
pow(…)
    pow(x, y)
    Return x**y (x to the power of y).
```

Warning

- Some programming languages allow the protection of some variables, making them constants (non mutable)
- Python doesn't have that mechanism
- This is possible:

```
>>> import math
 >>> math.pi = 5
```

- Python's philosophy:

"We are responsible adults"

Warning 2

- Do not name your program file using a module file NEVER

Checking the offer of a module

- Using `help` (we already know that one)
- Using the function `dir`
  - Shows only the variables and function names
  -
```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

builtins

`builtins` is the module with all the functions we've been calling without importing anything (e.g. `print`, `input`, `len`, `pow`, ….)

- All members are automatically imported by Python

```
   from builtins import *
    import builtins as __builtins__
```

Just a guess.

- Checkout what is offered by `__builtins__`
  - Many of the members available are used to signal errors
  - Those error signal items are also types (like `int`, `float`, `str`, etc…)

Own modules

- How to create:
  - Write the functions in a text file (e.g. using IDLE)
  - Save it with suffix `.py` (e.g. `physics.py`)
- How to use:
  - From IDLE Shell, just import (e.g. `>>> import physics`)
  - From another source code file, just import (top of the file)

- Some notes:
  - The name of the module should reflect what the available functions are about
  - Functions inside the module should be related

How Python finds modules

- Python looks for modules in the current working directory and in a series of other predefined directories
- Those directories are defined in a system variable called `path`
- Let's look at it:

```
>>> import sys
  >>> print(sys.path)
  ['', '/home/aanjos', '/usr/bin',
  '/usr/lib/python35.zip', '/usr/lib/python3.5',
  '/usr/lib/python3.5/plat-x86_64-linux-gnu',
  '/usr/lib/python3.5/lib-dynload',
  '/usr/local/lib/python3.5/dist-packages',
  '/usr/lib/python3/dist packages']
```

Using our modules

- Our modules/files will be saved in some folder
- We could modify the path variable to include that folder
- We are not going to do that
- Instead, we are going to place the module we want to import in the same directory as the file that imports it (or vice-versa)

Our module

- We already have a module with several functions

  `celsius_to_fahrenheit`

  `fahrenheit_to_celsius`

- Every time we do “Run Module” on IDLE, the shell:
  - Changes its working directory to the directory where the file is (e.g. `/home/aanjos/CT/week06`)
  - Imports the module (e.g. `converter`)

4 minute problem

- Write a program that tells which temperature is greater
- It must work even if temperatures are in different units
- The allowed units are Celsius and Fahrenheit
  - **Copy** the module to the same folder of your new program
  - Import it from your new program

- Example of the program running:

```
Temperature 1: 20
Unit: C
Temperature 2: 30
Unit: F
Temperature 1 is greater
```

"Loose" source code

- If there is code testing the function of the module (inside the module), that code is executed when the module is imported
- We don't, usually, want that
  - We just want the functions of the module available to our new program

Preventing to run “loose” code

- Python defines a special internal string variable that “knows” if a module is being imported or ran directly
- That variable is called: `__name__`
- The value of `__name__` will be
  - The **name of the module** if the module was imported
  - The string **`"__main__"`** if the module was called directly
- In short: `__name__` will be **`"__main__"`** only in the “file” that is called directly

Trick to protect “loose” code

- As simple as:

```
if __name__ == "__main__":
    run_stuff()
    run_more_stuff()
    print('Bye…')
```

- Very useful to allow the inclusion of code to test the modules

Back to the questions

- What is a module?
- What are some of the Python standard modules?
- How can we make a module available to our program?
- How can we access the members of a module?
- How can we create our own modules?
- How can we prevent “loose” code to run upon importation of modules?

WESTMINSTER
International University in Tashkent

Further reading

- PP, chapter 4

WESTMINSTER
International University in Tashkent