

Lecture 3

Modules & Functions

Netids That Did Not Do the Quiz

- aal59
- abr75
- ajf235
- al728
- alb383
- ank43
- apj33
- awg68
- bhw44
- cdj44
- cfw56
- cjm279
- cvi3
- dg488
- djm438
- drh234
- ds653
- ech96
- efo5
- egm58
- gbf22
- gd243
- gem67
- gj54
- hc655
- hw386
- hy388
- iam9
- jbm247
- jc2543
- jjm448
- jl2879
- jrn56
- jt566
- jtk53
- jw834
- ksk75
- kt429
- lap248
- mdw97
- meb327
- mjs624
- mmb299
- mp723
- mrr87
- nhm42
- njf53
- otb6
- pwh37
- sar259
- sec269
- sk2448
- sr688
- srh78
- vkm22
- wpa26
- wta24
- xl237

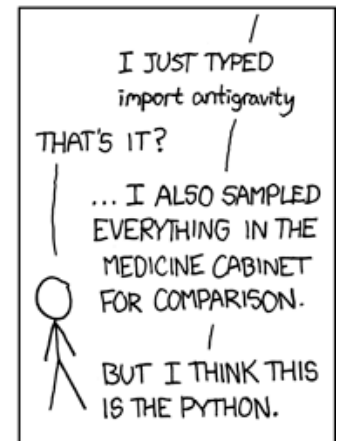
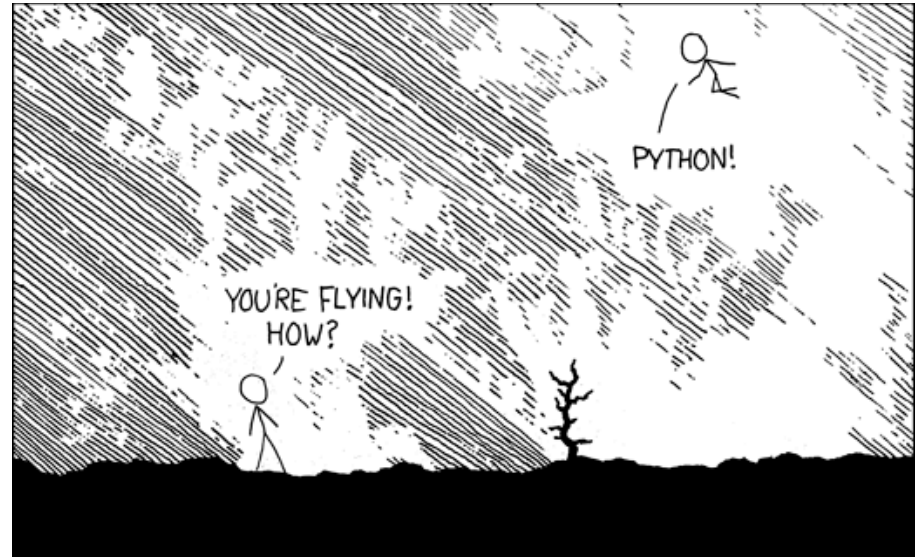
Readings for Next Two Lectures

This Week

- Sections 3.1-3.4, 3.13
- Browse the Python API
 - Do not need to read all of it
 - Look over built-in functions
 - Some interesting modules: `math`, `str`, and `sys`

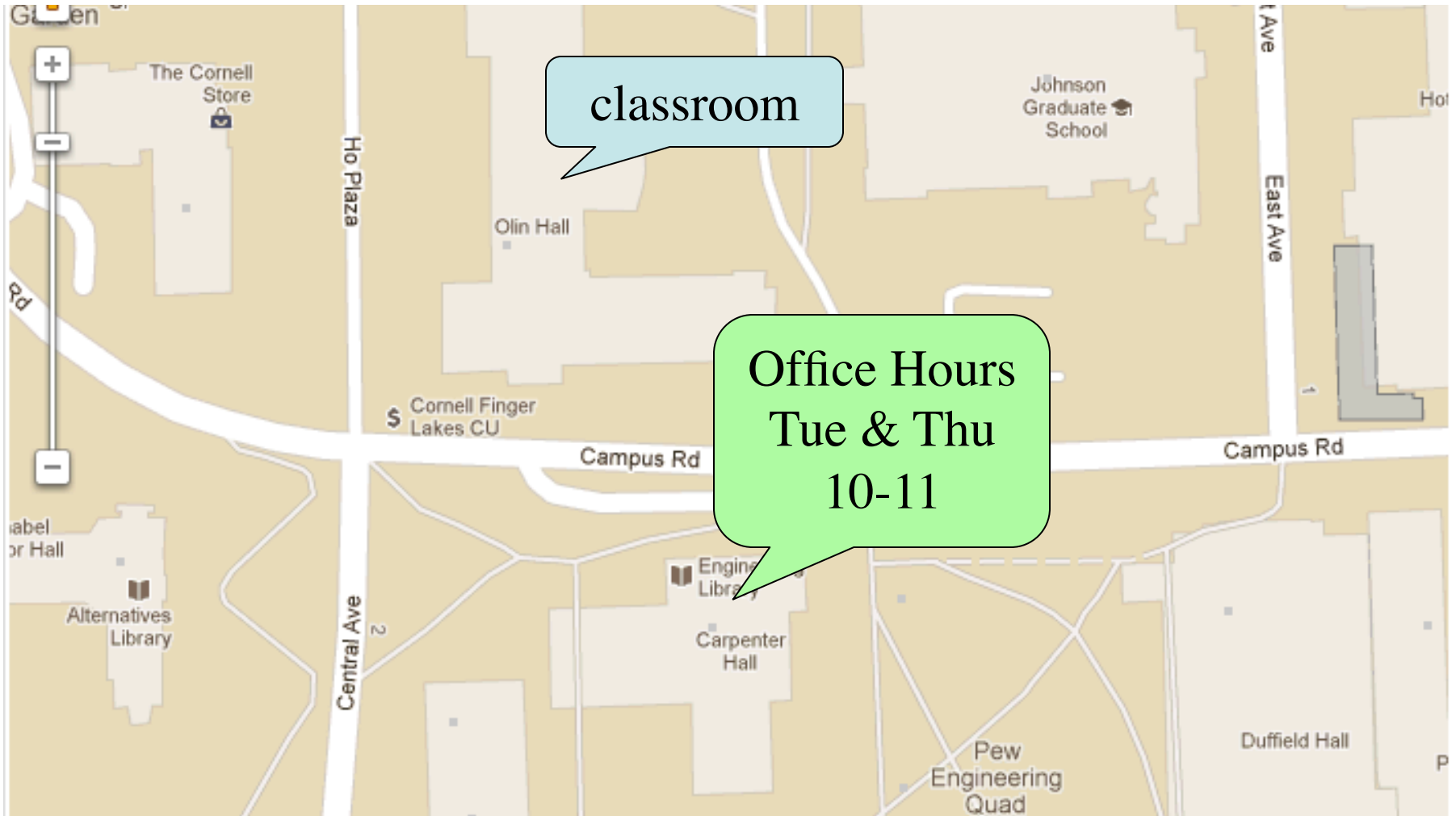
Next Week

- Sections 8.1, 8.2, 8.4, 8.5, 8.8



[xkcd.com]

Office Hours this Semester



Variables and Types

- Python is a **dynamically typed language**
 - Variables can hold values of any type
 - Type of value in variable can change over time
- The following is acceptable in Python:

```
>>> x = 1
```

 ← x contains an **int** value

```
>>> x = x / 2.0
```

 ← x contains a **float** value (why?)
- Alternative is a **statically typed language**
 - Each variable restricted to values of just one type
 - Java is an example of such a language

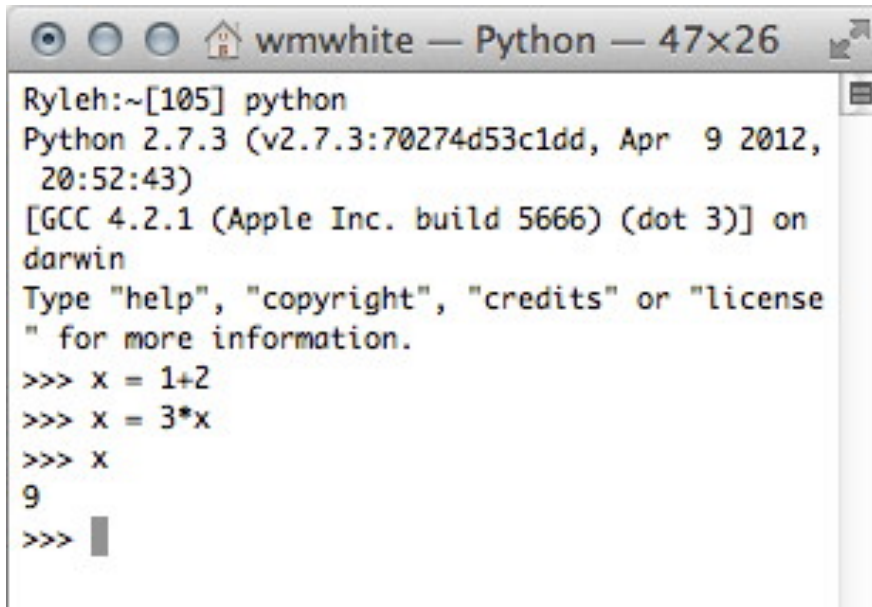
Dynamic Typing

- Often want to track the type in a variable
 - What is the result of evaluating x / y ?
 - Depends on whether x, y are **int** or **float** values
- Use expression `type(<expression>)` to get type
 - `type(2)` evaluates to `<type 'int'>`
 - `type(x)` evaluates to type of contents of x
- Can use in a boolean expression to test type
 - `type("abc") == str` evaluates to **True**

Eliminating Variables

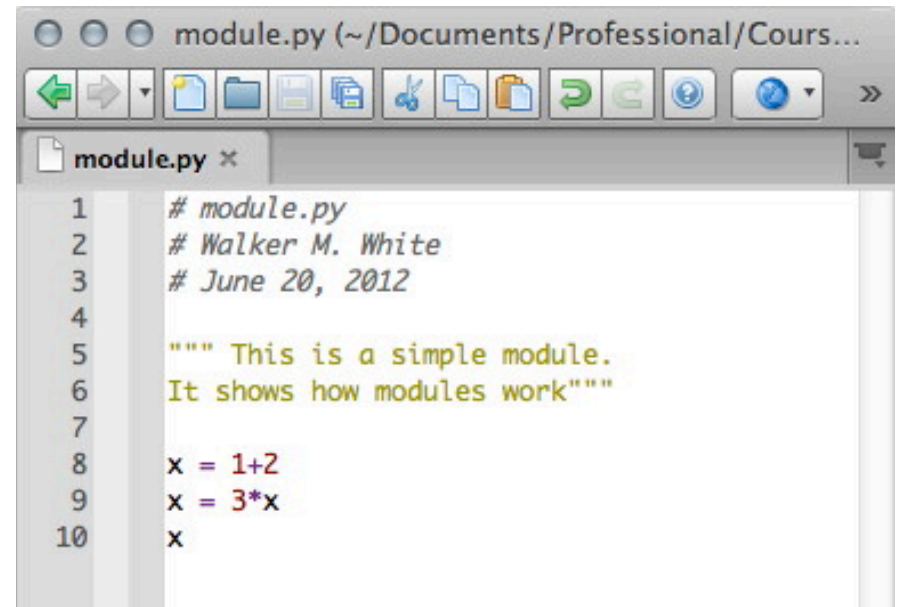
- Sometimes want to get rid of a variable
 - Not for performance; not a focus in Python
 - Do it to make code **cleaner/safer**
 - If refer to wrong variable, better **it not exist** (why?)
- Command: **del** *<variable>*
 - Variable ceases to exist
 - Expressions with variable will cause errors
 - Needs an assignment to exist again

Python Shell vs. Modules



```
Ryleh:~[105] python
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr 9 2012,
20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on
darwin
Type "help", "copyright", "credits" or "license
" for more information.
>>> x = 1+2
>>> x = 3*x
>>> x
9
>>>
```

- The interactive shell
 - Simple to use
 - Experience with Lab 1
- But very inefficient
 - One command at a time
 - What if need 1000+ lines?



```
1 # module.py
2 # Walker M. White
3 # June 20, 2012
4
5 """ This is a simple module.
6 It shows how modules work"""
7
8 x = 1+2
9 x = 3*x
10 x
```

- Alternative: **modules**
 - Files with commands
 - Write in a special editor
- Run module with **import**
 - Loads the file into Python
 - Executes each line

Using a Module

Module Contents

```
# module.py
```

Single line comment
(not executed)

```
""" This is a simple module.  
It shows how modules work"""
```

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

```
x = 1+2  
x = 3*x
```

Commands
executed on import

```
x
```

Not a command
Does nothing

Using a Module

Module Contents

```
# module.py
```

```
""" This is a simple module.  
It shows how modules work """
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

“**Module data**” must be
prefixed by module name

Prints **docstring** and
module contents

Python Shell

```
>>> import module
```

```
>>> x
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

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

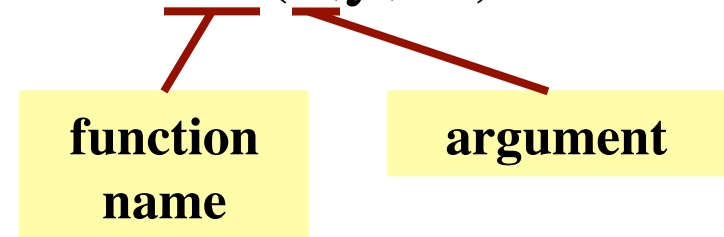
```
>>> module.x
```

```
9
```

```
>>> help(module)
```

Function Calls

- Python supports expressions with math-like functions
 - A function in an expression is a **function call**
 - Will explain the meaning of this later
- Function expressions have the form **fun(x,y,...)**



- Simplest example of functions are in module `math`

```
>>> import math
```

module variable

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

module function

Built-In Functions

- You have seen many functions already
 - Type casting functions: `int()`, `float()`, `bool()`
 - Dynamically type an expression: `type()`
 - Help function: `help()`
- Getting user input: `raw_input()`
- `print <string>` is **not** a function call
 - It is simply a statement (like assignment)
 - However, it is a function call in Python 3.x
 - Python 3.x: `print(<string>)`

Using the from Keyword

```
>>> import math
```

```
>>> math.pi
```

Must prefix with
module name

```
3.141592653589793
```

```
>>> from math import pi
```

```
>>> pi
```

No prefix needed
for variable pi

```
3.141592653589793
```

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

```
>>> cos(pi)
```

```
-1.0
```

No prefix needed
for anything in math

- Be careful using from!
- Modules are **namespaces**
 - There is only one variable or function for each name
 - Other modules may reuse names for variable/function
 - Prefix keeps them distinct
- **Example:** badpi.py

How Well Are You Following?

Module Contents

```
# data.py
```

```
""" Module with two variables """
```

```
x = 4
```

```
y = 3
```

A: 3

B: 7

C: 6 **CORRECT**

D: 4

E: I do not know

Python Shell

```
>>> x = 1
```

```
>>> y = 2
```

```
>>> from data import x
```

```
>>> x+y
```

```
???
```

How Well Are You Following?

Module Contents

```
# data.py
```

```
""" Module with two variables """
```

```
x = 4
```

```
y = 3
```

- A: 6
- B: 7 **CORRECT**
- C: Error!
- D: I do not know

Python Shell

```
>>> from data import *
```

```
>>> x = 3
```

```
>>> from data import x
```

```
>>> x+y
```

```
???
```

Importing a variable
“clobbers” any existing
variable of same name

Python Comes with Many Modules

- `io`
 - Read/write from files
- `math`
 - Mathematical functions
- `random`
 - Generate random numbers
 - Can pick any distribution
- `string`
 - Useful string functions
- `sys`
 - Information about your OS
- Complete list:
 - <http://docs.python.org/library>
 - **Library**: built-in modules
 - May change each release
 - Why version #s are an issue
 - Documentation is the **API**
 - Application
 - Programming
 - Interface
 - **Interface**: *specification* of the functions and data in a module

Reading the Python API

The screenshot shows the Python 2.7.3 documentation for the `math` module. The page title is "9.2. math — Mathematical functions". The main content area contains the following text:

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

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all

Return `x` with the sign of `y`. On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns `-1.0`.

Return the absolute value of `x`.

`math.factorial(x)`
Return `x` factorial. Raises `ValueError` if `x` is not integral or is negative.
New in version 2.6.

`math.floor(x)`
Return the floor of `x` as a float, the largest integer value less than or equal to `x`.

Callouts from the image:

- Function name**: Points to `math.ceil(x)`.
- Number of arguments**: Points to `x` in `math.ceil(x)`.
- What the function evaluates to**: Points to the description "Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`."

The Komodo Editor

The image shows a screenshot of the Komodo Editor interface. The main window displays a Python file named `hello.py` with the following code:

```
1 # hello.py
2
3 """ Simple module that prints out "Hello Word!" """
4
5 print "Hello World!"
```

Callouts point to various features:

- Line numbers:** A callout points to the line numbers 1 through 5 on the left side of the editor.
- Current working directory:** A callout points to the `modules` directory in the `Places` sidebar.
- Current active module:** A callout points to the `hello.py` tab in the top toolbar.
- Execution output when module is "run":** A callout points to the `Hello World!` output in the `Command Output` pane at the bottom.
- Run Python File button:** A callout points to the `Run Python File` button in the `Toolbox` on the right.
- See website for how to add button:** A callout points to the `Run Python File` button in the `Toolbox`.
- Tabs for open module files:** A callout points to the tabs for `badpi.py`, `data.py`, `module.py`, and `hello.py` in the top toolbar.

Lab Next Week

- Working with modules
 - Importing and accessing functions
 - Writing your own modules
- Working with the Python API
 - Reading function specifications
 - Using them properly
- Getting us ready for the first assignment
 - **Missing:** how to make your own functions

