# CSC148 Ramp-up
# Fall 2014

Michael Kimmins

(Based mostly on slides from Orion Buske)

(based on notes by Velian Pandeliev, Jonathan Taylor, Noah Lockwood, and software-carpentry.org)

# Overview

In the next 6 hours, we’ll cover the background required for CSC148.

This session is for students with programming experience who haven't necessarily taken the prerequisite, CSC108.

Please ask questions!

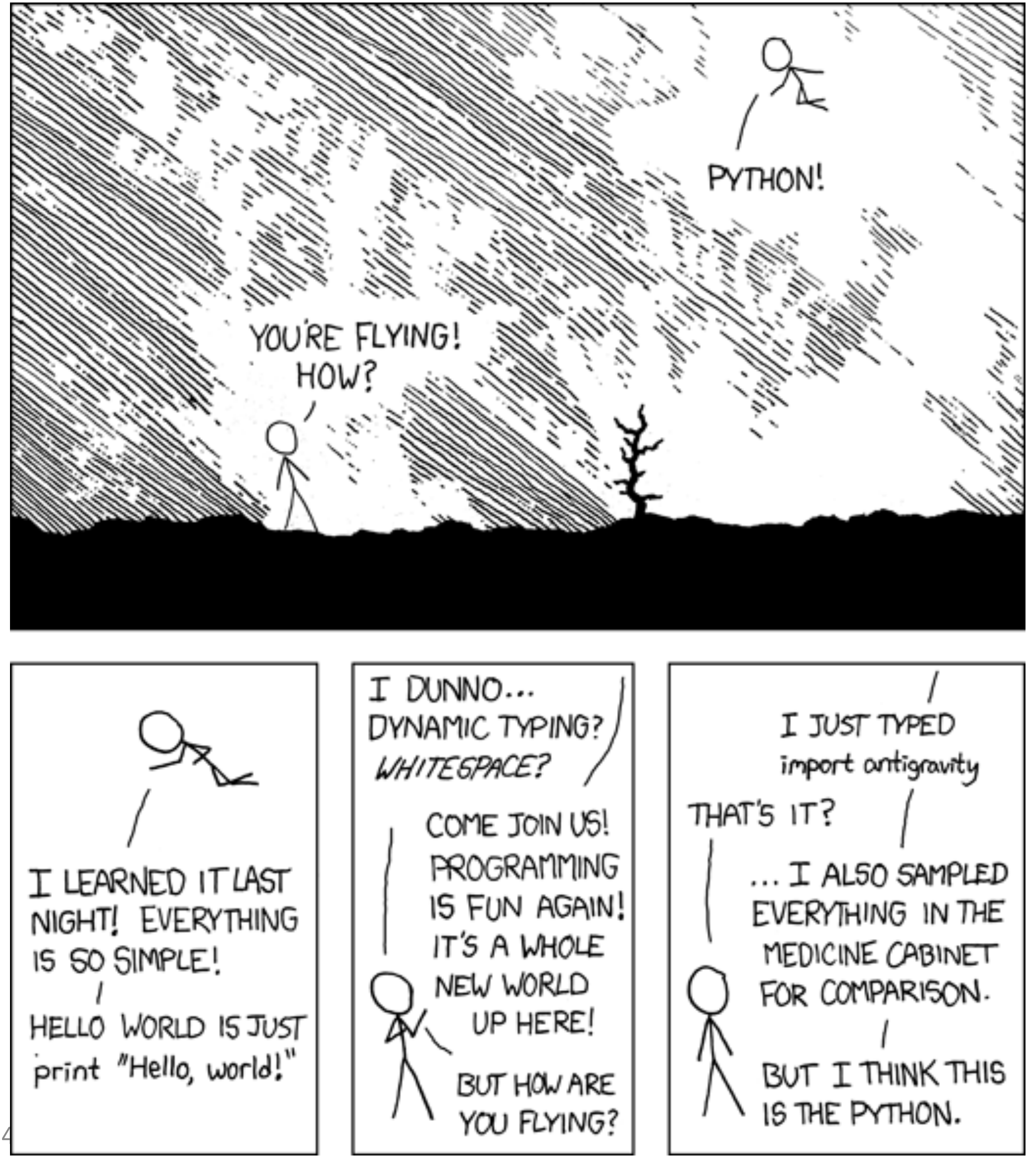
# Outline

- Talking
- Talking
- Talking
- Lunch
- Talking
- Talking
- Talking

# More explicit outline

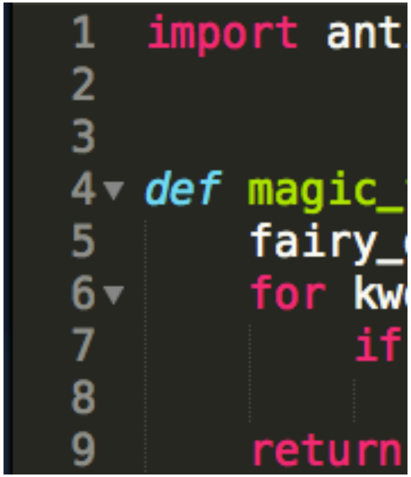
- Variables and types
- Lists, tuples, and for loops
- Conditionals and functions
- Lunch
- Dictionaries and files
- While loops and modules
- Classes and objects

# Meet Python...


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

# Whitespace matters

Python



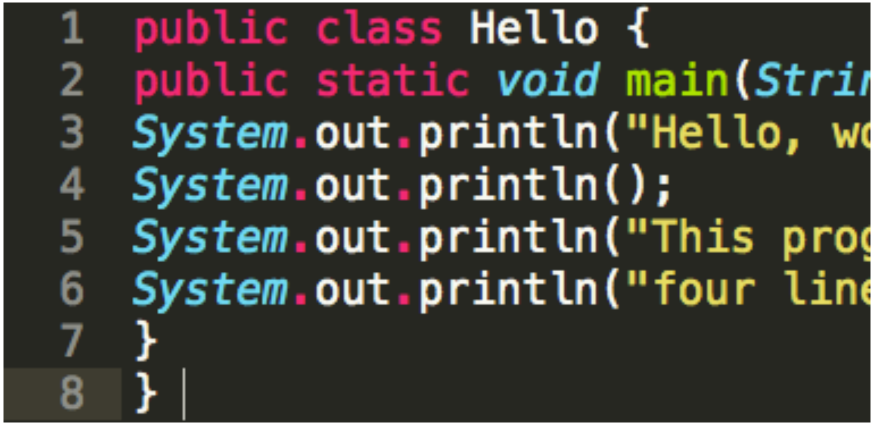
```
import ant


def magic_
    fairy_
    for kw
        if

    return
```

**vs**

Java



```
public class Hello {
public static void main(Stri
System.out.println("Hello, wo
System.out.println();
System.out.println("This prog
System.out.println("four line
}
}
```

# Let's speak some Python

- Python is **interpreted** (no compilation necessary)
- **Whitespace** matters (4 spaces for indentation)
- No end-of-line character (no semicolons!)
- No extra code needed to start (no "public static ...")
- Python is **dynamically typed** (a function can take multiple different types, have different behaviors)
- Python is **strongly typed** (all values have a type)

```
# Comments start with a '#' character.
# Python has dynamic typing, so:
x = 5  # assignment statement (no type specified)
x = "jabberwocky"  # re-assign x to a string
print(x)  # prints "jabberwocky"
```

# Let's speak some Python

- Python is **strongly typed**.

```
>>> def foo(x):
...     return x+1
...
>>> foo(1)
2
>>> foo("a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
TypeError: cannot concatenate 'str' and 'int'
objects
```

# Let's speak some Python

- Python is **dynamically typed**.

```
>>> def foo(x):
...     y = x + 1
...     z = y + "a"
...     return z
...
>>> foo (1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in foo
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

# Let's speak some Python

- Programmer must provide type safety

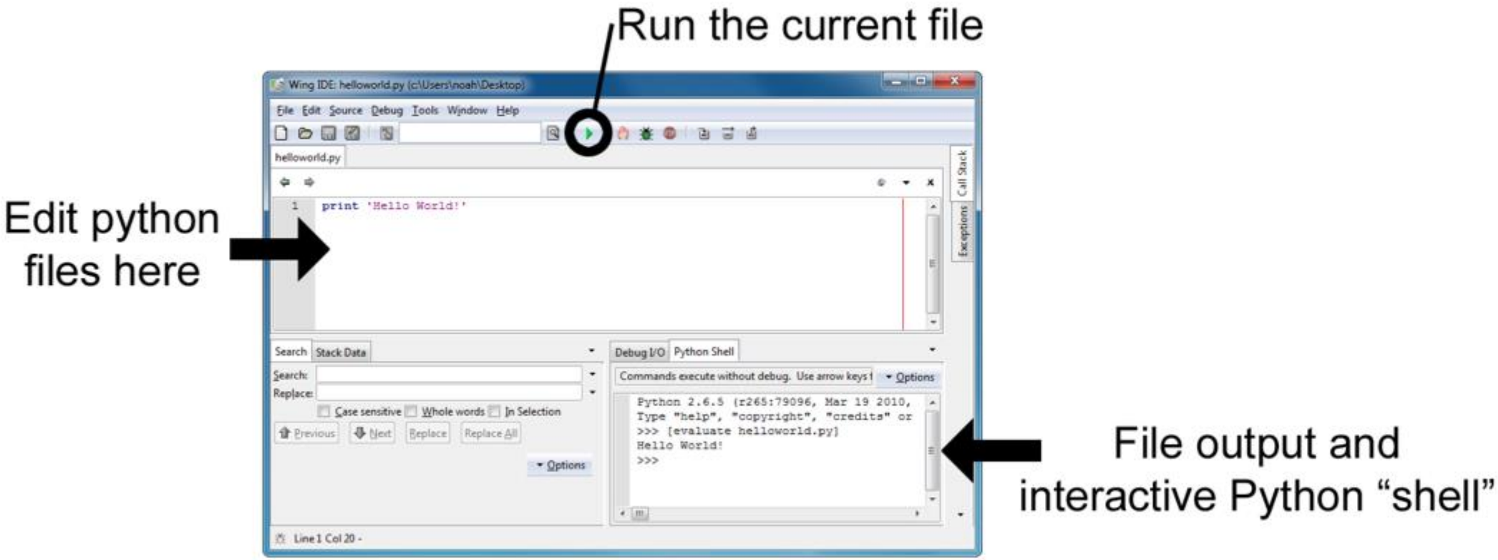
```
>>> def bar(x):
...     if type(x) == int:
...         print "This is an int."
...     else:
...         print "This is something else."
...
>>> bar(4)
This is an int.
>>>
>>> bar("4")
This is something else.
```

# Python programs

- Programs are stored in .py files
- From the command line (for teh hax0rz):

```
#user@redwolf:~$ python myfile.py
```

- Using the Wing IDE (Integrated Dev. Environment)



Run the current file

Edit python files here

File output and interactive Python “shell”

```
print 'Hello World!'
```

Commands execute without debug. Use arrow keys t

```
Python 2.6.5 (r265:79096, Mar 19 2010,
Type "help", "copyright", "credits" or
>>> [evaluate helloworld.py]
Hello World!
>>>
```

# The blueprint of a Python file:

```
from random import randint
from math import cos
```

import names from other modules

# The blueprint of a Python file:

```
from random import randint
from math import cos
```

import names from other modules

```
def my_function(arg):
    ...
    return answer

class MyClass:
    ...
```

define functions and classes

# The blueprint of a Python file:

```
from random import randint
from math import cos
```

import names from other modules

```
def my_function(arg):
    ...
    return answer

class MyClass:
    ...
```

define functions and classes

```
if __name__ == "__main__":
    my_variable = 21 * 2
    ...
```

your main block goes down here!

# The blueprint of a Python file:

```
from random import randint
from math import cos

def my_function(arg):
    ...
    return answer

class MyClass:
    ...

if __name__ == "__main__":
    my_variable = 21 * 2
    ...
```

**the main block mantra**

# The blueprint of a Python file:

```
from random import randint
from math import cos

def my_function(arg):
    ...
    return answer

class MyClass:
    ...

if __name__ == "__main__":
    my_variable = 21 * 2
    ...
```

note the case of different names and how we use whitespace

# Interactive Python

- Python can also be run interactively, from the bottom-right of Wing, or by typing `python` or `python3` on the command line.
- The result is automatically shown (unlike in a program, where you must call `print`).

```
#user@redwolf:~$ python
Python 3.2.3 (v3.2.3:3d0686d90f55, Apr 10 2012, 11:25:50)
Type "help", "copyright", "credits" or "license" for more
information.
>>> 42
42
>>> (2 ** 3 - 4) / 8
0.5
```

# Getting help

Official Python documentation:

`http://docs.python.org/py3k/library/`

The `help` function provides usage information:

```
>>> help(print)
```

The `dir` function shows names within a given type, module, object:

```
>>> dir(str)
```

# Moar resources!

Last term's 108 and 148 course websites:

`http://www.cdf.utoronto.ca/~csc108h/summer/`

`http://www.cdf.utoronto.ca/~csc148h/summer/`

Software Carpentry (online lectures):

`http://software-carpentry.org/`

Google!

`http://lmgtfy.com/?q=python+add+to+list`

# Learn you to good speak Python

Python's style guide:

`http://www.python.org/dev/peps/pep-0008/`

Google's Python style guide:

`http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`

Expert mode:

`pychecker: http://pychecker.sourceforge.net/`

`pyflakes: https://launchpad.net/pyflakes/`

# Variables (storing data)

Variables refer to an **object** of some **type**

Several basic data types:

- Integers (whole numbers): **`int`**

```
>>> the_answer = 42
```

- Floating-point (decimal) numbers: **`float`**

```
>>> pi = 3.14159
>>> radius = 2.0
>>> pi * (radius ** 2)
12.56636
```

5 / 2 -> 2.5     5 % 2 -> 1     5 // 2 -> 2

- operators:  `*   /   %   +   -   **   //`
- "shortcut" operators:  `x = x + 1` → `x += 1`

# More types (kinds of things)

- Boolean (True/False) values: **`bool`**

```
>>> passed = False
>>> not passed
True
>>> 5 < 4  # comparisons return bool
False
>>> 5 and 4  # this can bite you
4
```

- Operators: `and  or  not`

# More types (kinds of things)

- None (it's Python's NULL): **`NoneType`**

```
>>> x = None
>>> print(x)
None
>>> x
>>> type(x)
<type 'NoneType'>
>>> # Weird, we'll discuss this later
```

# Strings

Index 0 means the first letter.

- Strings (basically lists of characters): **`str`**

```
>>> welcome = "Hello, world!"
>>> welcome[1]    # index, starting with 0
'e'
```

- Slices return substrings:

```
>>> welcome[1:5] # slice with [start:end]
'ello'
>>> welcome[:3]  # start defaults to 0
'wel'
>>> welcome[9:]  # end defaults to None (wtf?)
'rld!'
>>> welcome[:-1] # index/slice with negatives
'Hello, world'
```

```
>>> welcome[:3]
'ld!'
```

# Working with strings

- Stick strings together (concatenation):

```
>>> salutation = "Hello, "
>>> name = "Orion"
>>> salutation + name  # evaluates to a new string
'Hello, Orion'
```

- The `len` function is useful:

```
>>> len(name)  # number of characters
5
```

# Tons of useful methods

- Here are some, look at `help(str)` for more:

```
>>> name = "Orion"
>>> name.endswith('ion')
True
>>> 'rio' in name   # substring testing
True
>>> name.startswith('orio')
???? Thoughts?
>>> name.lower()
'orion'   # new string!
>>> name.index('i')
2   # What did this do? Try help(str.index)
```

# POP QUIZ!

Write a boolean expression that evaluates to:

`True` if the variable `response` starts with the letter "q", case-insensitive, or

`False` if it does not.

```
def qStarter(word):
    return word.startswith('q') or word.startswith('Q')
```

(in CS lingo, we'd say: `True` iff (if and only if) the variable `response` starts with the letter "q", case-insensitive)

# POP QUIZ!

```
response.lower().startswith('q')
```

# A little more on strings

- Strings are **immutable**, meaning they can't be changed once created

  ```
  >>> name = 'Orion'
  >>> name[1] = 'n'
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: 'str' object does not support item
  assignment
  ```

- Empty strings are OK:

  ```
  >>> what_i_have_to_say = ''
  ```

# Making strings pretty

- String formatting (`str.format`):
  - http://docs.python.org/release/3.1.5/library/string.html#formatstrings
  - `{}` are replaced by the arguments to format
  - Formatting parameters can be specified using :`format`
    - Similar to printf

```
>>> n = 99
>>> where = "on the wall"
>>> '{} bottles of beer {}'.format(n, where)
'99 bottles of beer on the wall'
```

# Standard input/output

- Generating output (stdout): **`print()`**
  - Can take multiple arguments (will be joined with spaces)
- Reading keyboard input: **`input()`**

```
>>> name = input()
>>> name
'Orion'
>>> print("Hello " + name)
Hello Orion
>>> "Hello {}".format(name)
'Hello Orion'  # Why quotes here?
```

# Converting between types

- AKA: how to sanitize user input
- Functions: `int()`, `float()`, `str()`, `bool()`

```
>>> float('3.14')
3.14
>>> int(9 / 5)   # truncates
1
>>> float(3)
3.0
>>> str(3.14)
'3.14'
>>> '{:.4f}'.format(3.14159265358)
'3.1416'
```

# Converting between types

- Don't do anything silly:

```
  >>> int('fish')
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: invalid literal for int() with base
10: 'fish'
```

- And beware:

```
  >>> int('3.0')
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: invalid literal for int() with base
10: '3.0'
```

# Exercise 1: Temperature

C = (5 / 9) * (F - 32)

- Write a program that:
  - prompts the user for degrees in Fahrenheit
  - converts the number into Celsius
  - prints out the number in Celsius
    - to just 2 decimal places, if you dare

(You can assume the user enters a number)

# Exercise 1: Solution

Self-check: does your code work for 98.6?

```
# Read in the input     input
fahrenheit = float(~~raw_input~~("Input temperature (F): ”))

# Convert to Celsius
celsius = (5 / 9) * (fahrenheit - 32)

# Display the answer
print("Temperature is {:.2f} degrees C".format(celsius))
```

# Sequences, of, things!

There are two main kinds of sequences (things in an order) in Python:

- The **[mighty]** `list`
- The (humble,) `tuple`

# [Lists, of, things]

- Lists are a very important data structure in Python
- They are a **mutable sequence** of **any objects**

```
>>> colours = ['cyan', 'magenta', 'yellow']
>>> friends = []  # forever alone
>>> random_stuff = [42, 3.14, 'carpe diem']
>>> wtf = [[], [2, 3], friends]  # this is crazy
>>> my_friends = list(friends)  # copy a list
```

- Index and slice like strings:

```
>>> colours[0]        # indexing returns the element
'cyan'
>>> random_stuff[2:]  # slicing returns a sub-list
['carpe diem']
```

# [Lists, of, things].stuff()

- We can change, add, and remove elements from lists

```
>>> marks = [98, None, 62, 54]
>>> marks[1] = 75  # change that None
>>> marks.append(90)   # add 90 to the end
>>> marks.remove(62)   # remove the 62
>>> marks.sort()   # sort in place
>>> print(marks)
??? Thoughts?
[54, 75, 90, 98]
```

# [Lists, of, things].stuff()

`list is mutable, string is not.`

- Lots of other awesome features, too

```
>>> marks = [74, 62, 54]
>>> len(marks)   # size
3
>>> 54 in marks   # membership testing
True
>>> marks.pop(1)   # remove/return val at [~~2~~ 1]
62
>>> marks + [1, 2]   # concatenation
[74, 54, 1, 2]   # new list
```

# Variable aliasing

- Careful! Multiple variables might be referring to the **same** mutable data structure:

```
>>> sorted_list = [1, 2, 3]
>>> not_a_copy = sorted_list  # not a copy
>>> not_a_copy.append(0)
>>> sorted_list
[1, 2, 3, 0]  # crap

>>> actually_a_copy = list(sorted_list)
>>> another_copy = sorted_list[:]
```

# (Tuples, of, things)

- Tuples are like fast, simple lists, that are **immutable**

```
>>> stuff = (42, 3.14, 'carpe diem')
>>> stuff[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

- Can always create a list from them:

```
>>> L = list(stuff)
```

- >.<   A little weird to get a 1-element tuple:

```
('a') -> 'a'
('a',) -> ('a',)
```

# For loops! (you spin me right round baby…)

- **For loops** repeat some code for **each** element in a sequence
  - This is a foreach loop in most languages

```
>>> colours = ['red', 'green', 'blue']
>>> for colour in colours:
...     print(colour)
...
red
green
blue
```

# For loops! (you spin me right round baby...)

- But wait, I actually *wanted* the index!
  - Use **`range`**`(n)` in a for loop to loop over a range.

```
>>> for i in range(2):
...     print(i)
0
1
```

  - To start at a value other than 0:

```
>>> for i in range(4, 6):
...     print(i)
4
5
```

# For loops! (you spin me right round baby...)

- But wait, I actually *wanted* the index!
  - How should we loop over the indices of a list?

```
>>> for i in range(len(colours)):
...     print("{}. {}".format(i, colours[i]))
...
0. red
1. green
2. blue
```

# For loops! (you spin me right round baby...)

- But wait, I actually *wanted* the index!
  - Now, over the indices **and items**!

```
>>> n = len(colours)
>>> for (i, colour) in zip(range(n), colours):
...     print("{}. {}".format(i, colour))
...
0. red
1. green
2. blue
```

zip returns a list of pairs

# For loops! (you spin me right round baby...)

- But wait, I actually *wanted* the index!
  - Now, over the indices **and items**!

```
>>> for (i, colour) in enumerate(colours):
...     print("{}. {}".format(i, colour))
...
0. red
1. green
2. blue
```

# Exercise 2: Times table

Compute (and store in a variable) a times table for the numbers 0 through 9 as a **list of lists**.

For example, if it were just from 0 through 2, you should create:

```
[[0, 0, 0],
 [0, 1, 2],
 [0, 2, 4]]
```

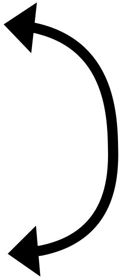
# Exercise 2: Solution

```
table = []
n = 10  # from 0 to (n - 1)
for i in range(n):
    # Compute the n'th row
    row = []
    for j in range(n):
        row.append(i * j)
    # Add row to full table
    table.append(row)
```

# Exercise 2: Solution

```
table = []
n = 10  # from 0 to (n - 1)
for i in range(n):
    # Compute the n'th row
    row = []
    # Add row to full table
    table.append(row)

    for j in range(n):
        row.append(i * j)
```



Does this still work?

# Exercise 2: Alternate solution

```
table = [[row * col for col in range(10)]
         for row in range(10)]
```

(list comprehensions FTW!)

# Conditionals (if, elif, else)

- **If statements** allow you to execute code sometimes (based upon some **condition**)
- **elif** (meaning 'else if') and **else** are optional

```
if amount > balance:
    print("You have been charged a $20"
          " overdraft fee. Enjoy.")
    balance -= 20
elif amount == balance:
    print("You're now broke")
else:
    print("Your account has been charged")

balance -= amount  # deduct amount from account
```

# Functions (basically the best things ever)

- They allow you to group together a bunch of statements into a block that you can call.
- "If you have the same code in two places, it will be wrong in one before long."
- "Never copy-paste code if at all possible."
- They can take in information (**arguments**) and give back information (**return value**).
- **Important**: If you don't specify a return value, it will be None

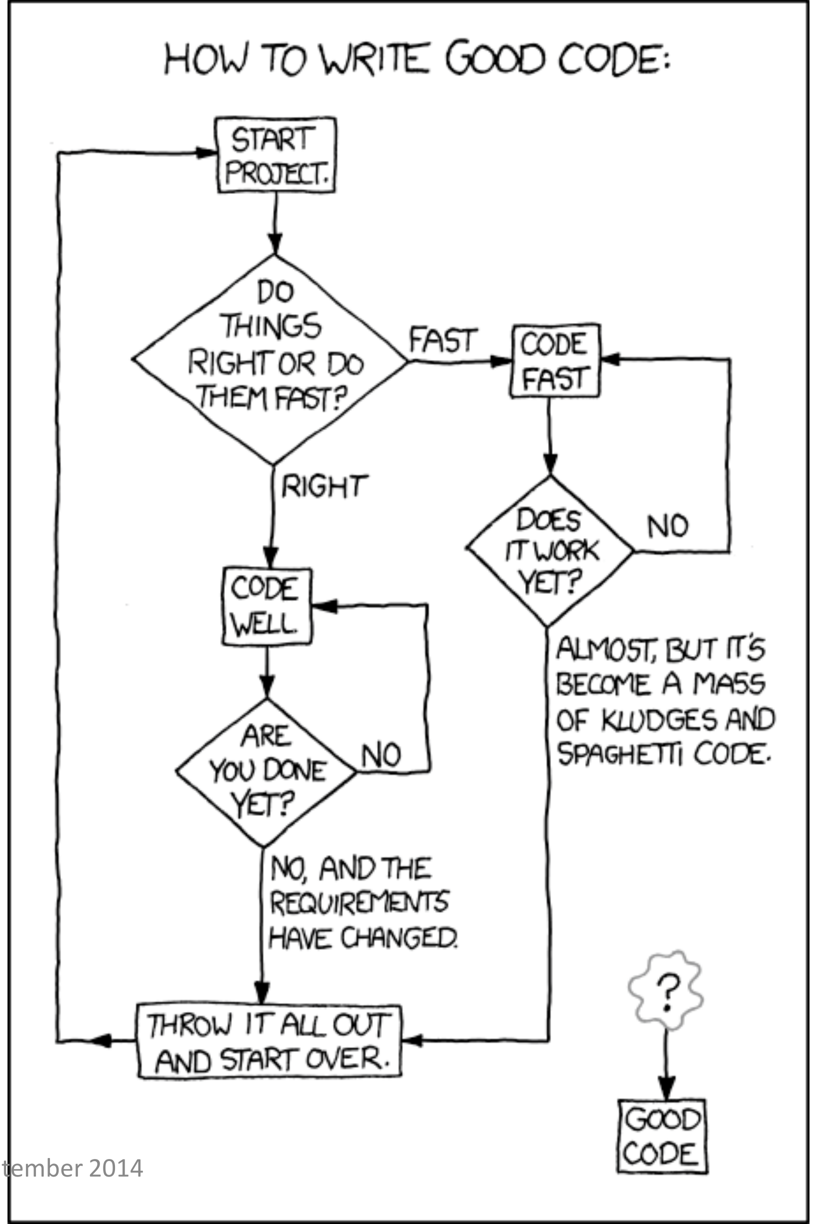
```
def celsius_to_fahrenheit(degrees):
    return (9 / 5) * degrees + 32

f = celsius_to_fahrenheit(100)
```

# Docstrings

- They should have a **docstring** (a multi-line, meaning triple-quoted, string right after the declaration)
- Describes **what** the function does, **not how** it does it.
- Describe the argument and return types.
- It is shown when **help** is called on your function, so it should be sufficient for other people to know how to use your function.

```
def celsius_to_fahrenheit(degrees):
    """(int or float) -> float
    Convert degrees from C to F.
    """
    return (9 / 5) * degrees + 32
```



HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
?
THROW IT ALL OUT AND START OVER.
GOOD CODE

**Very important.**

# Changing things

- Functions can modify mutable arguments

```
def double(L):
    """list -> NoneType
    Modify L so it is equivalent to L + L
    """
    for i in range(len(L)):
        L.append(L[i])

L = [1, 2, 3]
L = double(L)   # Don't do this! Why?
# double(L) changes the list and then returns None
```

# Changing things

- Functions can modify mutable arguments
- **If no return is specified, None is returned**

```
def double(L):
    """list -> NoneType
    Modify L so it is equivalent to L + L
    """
    for i in range(len(L)):
        L.append(L[i])

L = [1, 2, 3]
double(L)
print(L)  # [1, 2, 3, 1, 2, 3]
```

# Changing things

```
Immutable:
>>> stuff = (42, 3.14, 'carpe diem')
>>> stuff[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment
```

# Changing things

Immutable:

```
>>> hi = "hello"
>>> hi[0]
'h'
>>> hi[0] = "j"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support
item assignment
```

# Changing things

```
Immutable:
>>> a[0]
1
>>> a[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
item assignment
```

# Changing things

mutable:

```
>>> list = [1,2,3,4]
>>> list[0]
1
>>> list[0] = 10
>>> list
[10, 2, 3, 4]
```

# More control tools

- Break: “break out” of the loop. Think of breaking from a prison
- Pass: A null operation. Nothing happens

```
>>> def do_something(number):
...     for index in range(number):
...         if number == 2:
...             break
...         if number == 3:
...             pass
...         else:
...             print(number)
...     return None
```

# Exercise 3: Functions

Two words are a reverse pair if each word is the reverse of the other.

1) Write a function `is_reverse_pair(s1, s2)` that returns `True` iff `s1` and `s2` are a reverse pair.

2) Write a function `print_reverse_pairs(wordlist)` that accepts a list of strings and prints out all of the reverse pairs in the given list, each pair on a line.

# Exercise 3: Solution

```
def is_reverse_pair(s1, s2):
    """(str, str) -> bool"""
    pass
```

# Exercise 3: Solution

```
def is_reverse_pair(s1, s2):
    if len(s1) != len(s2):
        return False

    for i in range(len(s1)):
        if s1[i] != s2[len(s2) - 1 – i]:
            return False

    return True
```

Or, using slicing:

```
def is_reverse_pair(s1, s2):
    return s1[::-1] == s2
```

# Exercise 3: Solution

```
def print_reverse_pairs(wordlist):
   """list -> NoneType"""
    pass
```

# Exercise 3: Solution

```
def print_reverse_pairs(wordlist):
    for s1 in wordlist:
        for s2 in wordlist:
            if is_reverse_pair(s1, s2):
                print("{}, {}".format(s1, s2))
```

# Eat ALL the things…

# {'dictionaries': 'awesome'}

- **Dictionaries** (type **`dict`**) are an **unordered** association of **keys** with **values**
- We usually use them to store associations:
  - like name -> phone number
  - phone number -> name
  - student id -> grade

# {'dictionaries': 'awesome'}

- **Dictionaries** (type **`dict`**) are an **unordered** association of **keys** with **values**
- We usually use them to store associations:
  - like name -> phone number
  - phone number -> name
  - student id -> grade
  - grade -> student id **#BAD, why?**

# {'dictionaries': 'awesome'}

- **Dictionaries** (type `dict`) are an **unordered** association of **keys** with **values**
- We usually use them to store associations:
  - like name -> phone number
  - phone number -> name
  - student id -> grade
  - grade -> list of student ids
- Keys must be **unique** and **immutable**

```
Keys are immutable.
```

# {'dictionaries': 'awesome'}

```
>>> scores = {'Alice': 90, 'Bob': 76, 'Eve': 82}
>>> scores['Alice']  # get
90
>>> scores['Charlie'] = 64  # set
>>> scores.pop('Bob')  # delete
76
>>> 'Eve' in scores  # membership testing
True
>>> for name in scores:  # loops over keys
...     print("{}: {}".format(name, scores[name]))
...
Charlie: 64
Alice: 88
Eve: 82
```

# A brief detour to open some files

- The naïve way:

```
f = open("myfile.txt")
for line in f:
  ...  # do something with each line
f.close()

# What happens if an error occurs?
```

# A brief detour to open some files

- Use `with/as` to open something for a while, but always close it, even if something goes wrong.

- Easiest way to read files:

```
with open('myfile.txt') as open_file:
  for line in open_file:
    ... # do something with each line
```

- Easiest way to write to files:

```
with open('myfile.txt', 'w') as open_file:
  print(data, file=open_file)  # write to the file
```

# A brief detour to open some files

- Writing

```
balance = 40
with open("output.txt", "w") as file:
    file.write("I can write\n")
    file.write("Account balance{}\n".format(balance))
```

```
with open('dog.txt', 'w') as open_file:
    print ('doggydog\nwoof', file=open_file)
import os
os.getcwd()
with open('dog.txt') as open_file:
    for line in open_file:
        print (line)

>>>doggydog

   woof
```

# A brief detour to open some files

Write a file `tolkien.txt` that takes a list, and writes down the given list entries to the file.

The function should take in this list:

```
>>> characters = ["Frodo Baggins", "Samwise Gamgee",
"Gandalf", "Aragorn II", "Legolas Greenleaf",
"Meriadoc Brandybuck", "Peregrin Took"]
```

# A brief detour to open some files

Write a file `tolkien.txt` that takes a list, and writes down the given list entries to the file.

The function should take in this list:

```
>>> characters = ["Frodo Baggins", "Samwise Gamgee",
"Gandalf", "Aragorn II", "Legolas Greenleaf",
"Meriadoc Brandybuck", "Peregrin Took"]
```

```
>>> with open("tolkien.txt", "w") as file:
...     for name in characters:
...         file.write("{}\n".format(name))
...     file.close()
```

# A brief detour to open some files

Use the text file we made right now, read from the file `tolkien.txt` and store each line in a list `characters` within Python.

# A brief detour to open some files

Use the text file we made right now, read from a a file `tolkien.txt` and store each line in a list `characters`.

```
>>> with open("tolkien.txt") as file:
...     characters = file.readlines()
...     file.close()
...
>>> characters
['Frodo Baggins\n', 'Samwise Gamgee\n', 'Gandalf\n',
'Aragorn II\n', 'Legolas Greenleaf\n', 'Meriadoc
Brandybuck\n', 'Peregrin Took\n']
```

What happened?

# A brief detour to open some files

Use the text file we made right now, read from a a file `tolkien.txt` and store each line in a list `characters`.

```
>>> characters = []
>>> for line in open ('tolkien.txt'):
...     characters.append(line.strip())
...
>>> characters
['Frodo Baggins', 'Samwise Gamgee', 'Gandalf',
'Aragorn II', 'Legolas Greenleaf', 'Meriadoc
Brandybuck', 'Peregrin Took']
```

Better.

# Exercise 4: Dictionaries

Write a function `print_record` that takes a dictionary as input. Keys are student numbers (`int`), values are names (`str`). The function should print out all records, nicely formatted.

```
>>> record = {1234: 'Tony Stark', 1138: 'Steve Rogers'}
>>> print_record(record)
Tony Stark (#1234)
Steve Rogers (#1138)
```

Write a function `count_occurrences` that takes an open file as input, and returns a dictionary with key/value pairs of each word and the number of occurrences of that word. (a word is a white-space delimited token, and can have punctuation)

```
>>> open_file = io.StringIO('a b a a c c a.')
>>> count_occurences(open_file)
{'a': 3, 'b': 1, 'a.': 1, 'c': 2}
```

hints: **`in`** and **`str.split`**

# Exercise 4: Solution

```
def print_record(record):
    """dict -> NoneType"""
    for pin in record:
        print('{} (#{})'.format(record[pin], pin))
```

# Exercise 4: Solution

```
def count_occurrences(file):
    """file -> dict"""
    counts = {}
    for line in file:
        for word in line.split():
          if word in counts:
              counts[word] += 1
          else:
              counts[word] = 1
    return counts
```

# While loops (right round right round...)

- **While loops** keep repeating a block of code while a condition is `True`

```
# What does this code do?
val = 10
while val > 0:
    print("hello")
    val -= 1

# prints "hello" 10 times
```

# While loops (right round right round...)

- **While loops** keep repeating a block of code while a condition is `True`

```
# What does this code do?
val = 167
while val > 0:
    if val % 2 == 0:
        print("0")
    else:
        print("1")
    val = int(val / 2)

# prints (reverse) binary representation of val
```

# While loops (right round right round...)

- **break** can be used to exit a loop early

```
# What does this code do?
while True:  # This is an infinite loop
    # Stop when the user types 'quit', 'Q', etc.
    response = input("Enter number or 'quit':")
    if response.lower().startswith('q'):
        break  # This breaks out of the loop

    ...
```

# Modules (why reinvent the wheel?)

Python has a spectacular assortment of **modules** that you can use (you have to import their **names** first, though)

```
>>> from random import randint  # now we can use it!
>>> randint(1, 6)  # roll a die
4  # http://xkcd.com/221/
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.cos(0)
1.0
>>> import datetime
>>> dir(datetime)
```

# Exercise 5: Guessing game

Implement a guessing game:

```
Guess a number between 0 and 100: 50
Too high.
Guess a number between 0 and 100: 25
Too low.
Guess a number between 0 and 100: 40
Too low.
Guess a number between 0 and 100: -2
Guess a number between 0 and 100: 47
Correct. Thanks for playing!
```

hint: **"random"** module

# Exercise 5: Solution

```
from random import randint
# Choose a random number
low = 0
high = 100
answer = randint(low, high)

found = False
while not found:
    print("Guess a number between {} and {}: "
          "".format(low, high), end="")
    guess = int(input())
    # Print response if guess is in range
    if guess >= low and guess <= high:
        if guess > answer:
            print("Too high.")
        elif guess < answer:
            print("Too low.")
        else:
            print("Correct. Thanks for playing!")
            found = True  # Or you could use break here
```

# Classes and objects - philosophy

- Classes are descriptions of types of things (like a blueprint), and objects are specific instances of a type (like the actual building).
- Objects have associated state (attributes) and behavior (methods).
- We usually want to hide the implementation as much as possible, so that the people using our classes don't need to know how they are implemented, and so they don't go mucking around where they shouldn't.
- These will be discussed in much more detail in 148.

# Classes and objects - simple e.g.

```
class Point:
    pass



# Then we can make a Point object and use it!
position = Point()
position.x = 5  # add attributes to our object
position.y = -2
print((position.x, position.y))  # (5, -2)
```

# Classes and objects - simple e.g.

```
class Point:
    """A new Point"""
    def __init__(self, x=0, y=0):
        """(Point, int, int) -> NoneType"""
        self.x = x
        self.y = y

# Then we can make a Point object and use it!
position = Point(0, 0)  # or Point(), since defaults
position.x += 5  # adjust the attribute values
position.y -= 2
print((position.x, position.y))  # (5, -2)
```

# Classes and objects - simple e.g.

```
    def translate(self, dx, dy):
        """(Point, int, int) -> NoneType
        Translate the point by dx and dy"""
        self.x += dx
        self.y += dy

# Then we can make a Point object and use it!
position = Point(0, 0)  # or Point(), since defaults
position.translate(dy=-2, dx=5)  # use keyword arguments
print((position.x, position.y))  # (5, -2)
```

# Classes and objects - simple e.g.

```
class Point:
    """A new Point """
    def __init__(self, x=0, y=0):
        """(Point, int, int) -> NoneType"""
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """(Point, int, int) -> NoneType
        Translate the point by dx and dy"""
        self.x += dx
        self.y += dy

    def __str__(self):
        """(Point) -> str"""
        return "({}, {})".format(self.x, self.y)

position = Point(5, -2)
print(position)  # (5, -2)
```

# Classes and objects - simple e.g.

```
def my_init(point, x=0, y=0):
    """(Point, int, int) -> NoneType"""
    point.x = x
    point.y = y

def my_translate(point, dx, dy):
    """(Point, int, int) """
    point.x += dx
    point.y += dy

class Point:
    pass

Point.__init__ = my_init
Point.translate = my_translate  # change the Point class

position = Point(2, 8)  # this works!
position.translate(5, -2)  # this works!
```

# Magic ~~mushrooms~~ methods

- "Magic" methods start and end with two underscores
- They allow your Classes to take advantage of Python built-ins and syntactic sugar, e.g.:

```
>>> my_object = MyClass()
>>> len(my_object)  # __len__
>>> str(my_object)  # __str__
>>> my_object[5]  # __getitem__
>>> for element in my_object:  # __iter__
```

# Magic ~~mushrooms~~ methods

```
class Point:
    def __init__(self, x=0, y=0):
        """(Point, int, int) -> NoneType"""
        self.x = x
        self.y = y
    def __str__(self):
        """(Point) -> str """
        return "({}, {})".format(self.x, self.y)
    def __repr__(self):
        """ (Point) -> str """
        return "Point({}, {})".format(self.x, self.y)
```

# Magic ~~mushrooms~~ methods

```
>>> p = Point(5, 3)
>>> str(p)
'(5, 3)'
>>> repr(p)
'Point(5, 3)'
>>> print(p)
(5, 3)
>>> p
Point(5, 3)
```

- `print` uses `str`
- the prompt uses `repr`

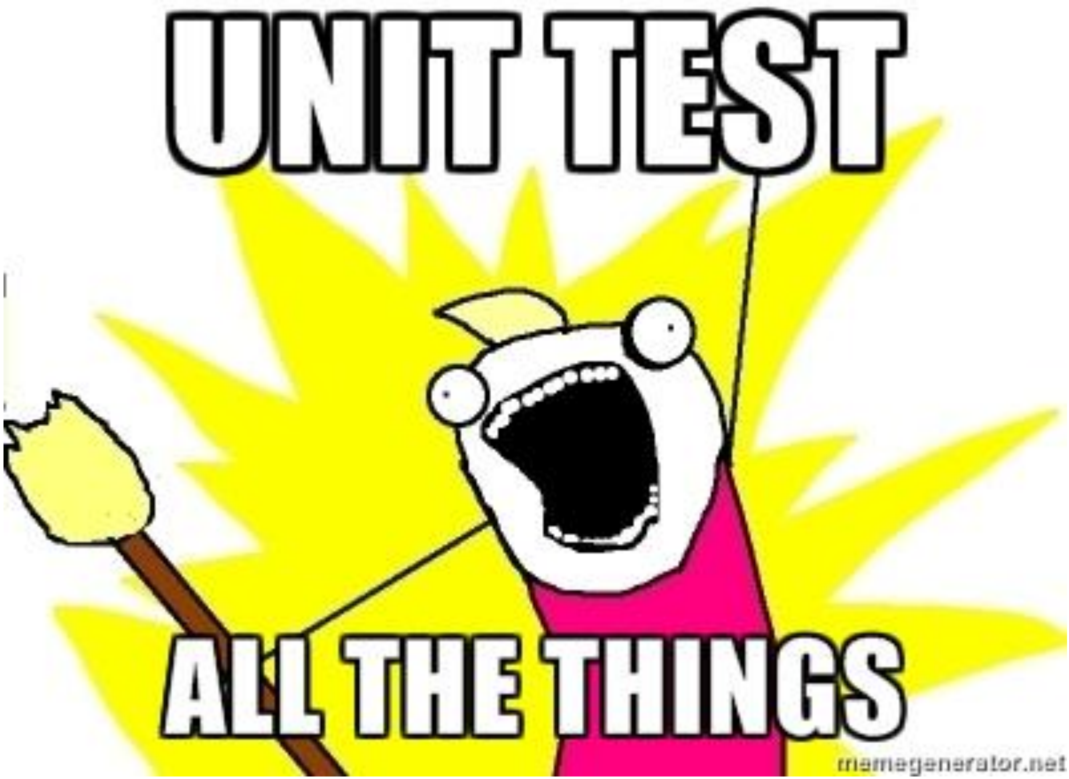
# Magic ~~mushrooms~~ methods

Remember this?

```
>>> x = None
>>> print(x)
None
>>> x
>>> # Weird, we'll discuss this later
```

- `print` uses `str`
- the prompt uses `repr`

# Testing the code



# Testing the code

- Why test?

  Assures correctness of the program under specific conditions

  Thinking of testing while coding makes the coder design a code that is better designed

  Helps you think about edge cases (e.g. What if user tries to delete a file that isn’t there? What if a function that takes mutable data is given an immutable type?)

# Testing the code

- even.py

```
def is_even(num):
""" (int) -> bool
 return True if num is even"""
    return number % 2 == 0
```

# Testing the code

Docstrings omitted for space!

Test_even.py

```
import unittest


class EvenTestCase(unittest.TestCase):
    def test_is_two_even(self):
        self.assertTrue(is_even(2))
if __name__ == '__main__': unittest.main()
```

# Testing the code

- In unit testing, test_* methods are recognized by the module.
- Also, setUp and tearDown are special methods

# Testing the code

Test_even.py

```
import unittest


class EvenTestCase(unittest.TestCase):
"""Tests for `even.py`."""
    def setUp(self):
        pass
    def tearDown(self):
        pass
if __name__ == '__main__': unittest.main()
```

# Classes and objects - complex e.g.

As a user of the IPhone class, we usually don't want to know what goes on under the surface. And Apple certainly doesn't want us messing around with what's inside (we might screw things up!). So this is how we might think about a class as a client:

```
class IPhone:
    def __init__(self):
        """Initialize the iPhone"""
        ...
    def call(self, phone_number):
        """Call the given phone number"""
        ...
    def kill_switch(self, authorization_code):
        """Brick the iPhone"""
        ...

# Then we can make an IPhone object and use it!
precious = IPhone()
precious.call('123.456.7890')
```

# Classes and objects - complex e.g.

As a developer, we want to hide the implementation as much as possible. This lets us change our implementation later without breaking everything!

```
class IPhone:
    def __init__(self):
        """Initialize the iPhone"""
        # Private attributes start with an underscore "_"
        self._call_timer = 0
        self._recent_calls = []
        self._network = RogersNetwork(self)

    def call(self, phone_number):
        """Call the given phone number"""
        self._recent_calls.append(phone_number)
        self._network.connect(phone_number)
```

# Exercise 6: NumberList

Write a class that stores a list of integers/floats and provides the following methods:

`sum()` - return the sum of the numbers in the list

`mean()` - return the average of the numbers in the list as a float

`min()/max()` - return the minimum/maximum element

`num_unique()` - return the number of unique elements in the list

For example:

```
>>> nl = NumberList([1, 2, 5, 1, 4, 3, 3])
>>> nl.sum()
19
>>> nl.num_unique()
5
```

**Hint:** Use the in keyword:

```
>>> nums = [1, 3, 9, 16]
>>> 3 in nums
True
>>> 7 in nums
False
```

# Exercise 6: Solution

```
class NumberList:
    def __init__(self, L):
        self._L = list(L)  # make a copy

    def sum(self):
        result = 0
        for value in self._L:
            result += value

        return result

    def mean(self):
        n = len(self._L)
        return self.sum() / n
```

# Exercise 6: Solution

```
...

def max(self):
    result = None
    for value in self._L:
        if result is None or x > result:
            result = x

    return result
```

# Exercise 6: Solution

```
...

def num_unique(self):
    # One of many possible solutions
    # Also: return len(set(self._L))

    seen = []
    for value in self._L:
        if value not in seen:
            seen.append(value)

    return len(seen)
```

fin