

CS3101 Programming Languages - Python

Spring 2010



Agenda

- Course description
- Python philosophy
- Getting started (language fundamentals, core data types, control flow)
- Assignment

Course Description

Instructor / Office Hours

- Josh Gordon
- PhD student
- Contact
 - joshua@cs.columbia.edu
- Office hours
 - By appointment, feel free to drop a line anytime



Syllabus

Lecture	Topics	Assignment
Jan 19 th	Language overview. Course structure. Scripting essentials.	HW1: Due Jan 26 th
Jan 26 th	Sorting. Parsing CSV files. Functions. Command line arguments.	HW2: Due Feb 2 nd
Feb 2 nd	Functional programming tools. Regular expressions. Generators / iterators.	HW3: Due Feb 9 th
Feb 9 th	Object oriented Python. Exceptions. Libraries I.	Project Proposal: Due Feb 16 th
Feb 16 th	GUIs. Databases. Pickling. Libraries II.	Course Project: Due Feb 28th
Feb 23 rd	Integration with C. Performance, optimization, profiling. Parallelization.	None

Grading

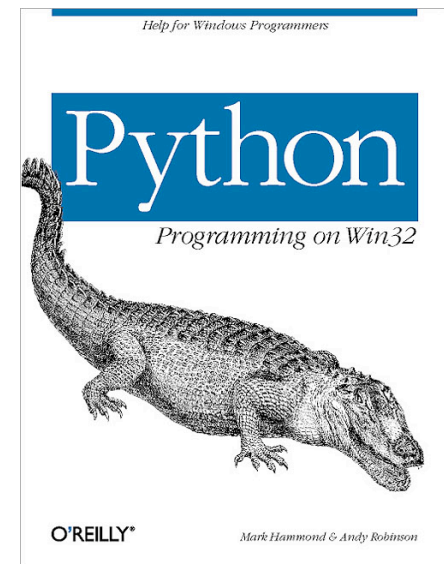
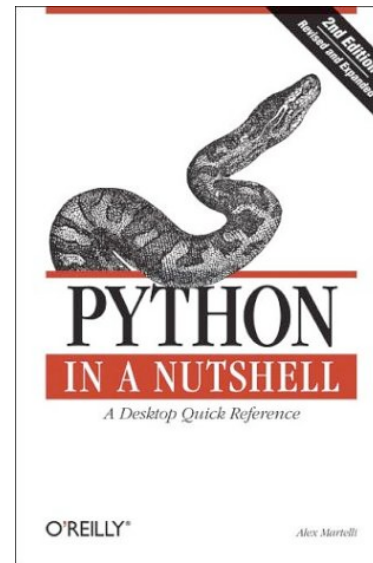
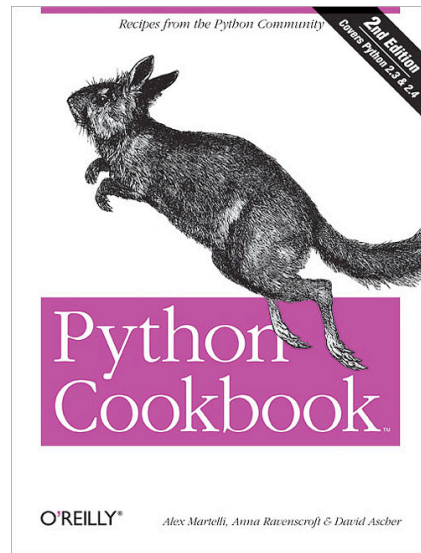
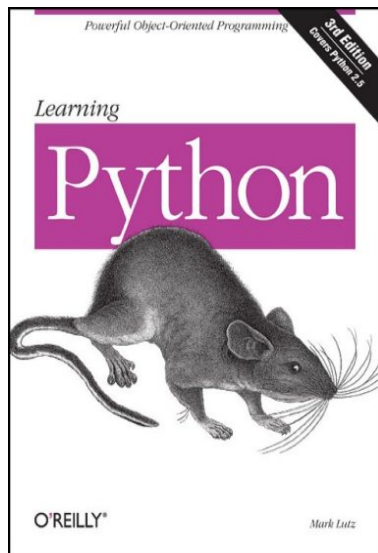
Assignment	Weight
Class participation	1/10
HW1	1/10
HW2	1/10
HW3	1/10
Proposal	1/10
Project	5/10
Extra credit challenge problems	Depends how far you get 😊

Late assignments: two grace days / semester, after which accepted at: -10% / day.

Resources / References

- Course website:
 - www.cs.columbia.edu/~joshua/teaching
 - Syllabus / Assignments / Slides
- Text books
 - Learning Python
 - Python in a Nutshell (available elect. on CLIO)
 - Python Cookbook
- Online doc:
 - www.python.org/doc

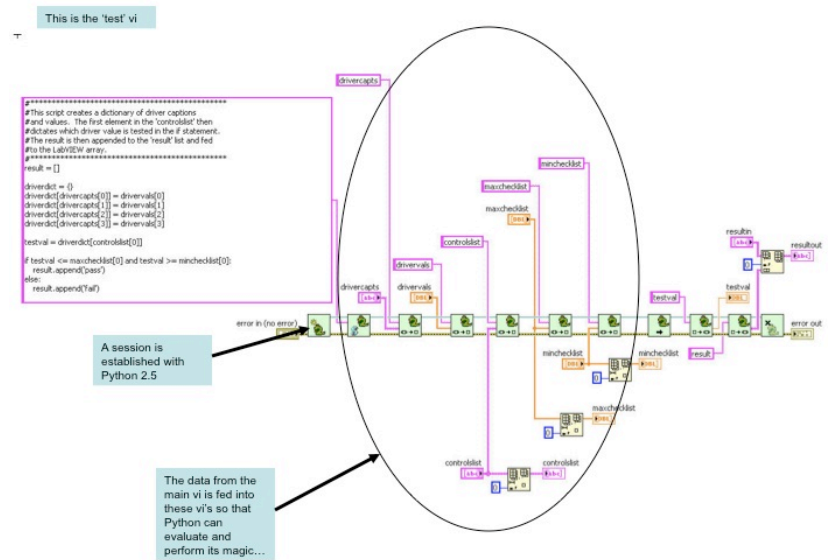
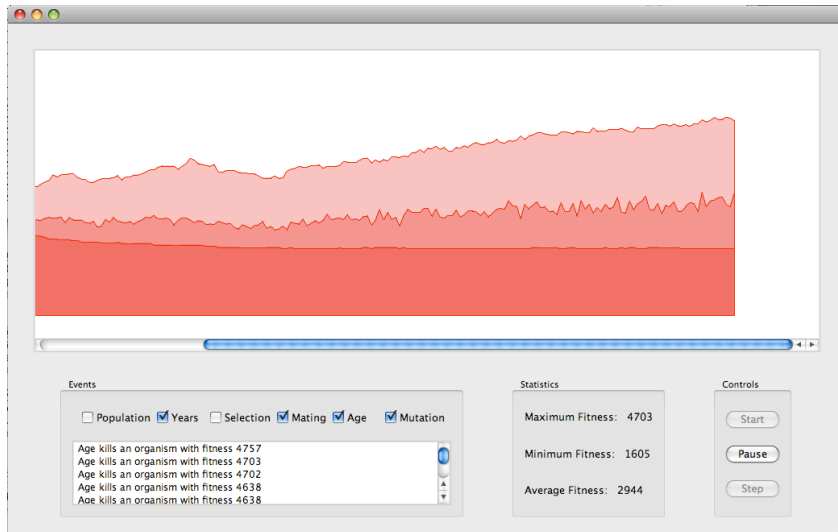
Ordered by technical complexity - notice anything?



Course Project

- Opportunity to leverage Python to accomplish something of interest / useful to you!
- Past projects:
 - Genetic algorithms to tackle NP-Hard problems
 - Solar system simulation via input to MAYA
 - Music recommendation system via mining Last.fm
 - Financial engineering utilities
 - Labview interface to control lab equipment
 - Sports scheduling
 - A webpage for elementary students

Previous Projects



Academic Honesty

- The Python community is top notch
- Incredible web resources
- Pitfalls:
 - Temptation to search for solutions
 - Learning only to concatenate other's work without thinking for yourself
- <http://www.cs.columbia.edu/education/honesty>

Python Philosophy

(I like to spend a lot of time on best practices)

What is Python?

- Powerful dynamic programming language
- Very clear, readable syntax (rejects the complexity of PERL in favor of a sparser, less cluttered grammar)
- Supports multiple programming paradigms (primarily object oriented, imperative, and functional)
- Dynamic type system (late binding)
- Automatic memory management / garbage collection
- Exception-based error handling
- Very high level built in data types
- Extensive standard libraries and third party modules
- Built from the ground up to be extensible via C (or Java, .NET)
- Embeddable within applications as a scripting interface

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than **right** now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Favorite Python Characteristics

- Emphasizes code readability / clarity
 - Key insight: code is read many more times than it is written
 - Allows the developer to focus on the problem domain rather than the implementation
 - Great for academic and research environments
- Batteries included
 - Powerful libraries for common tasks
 - Covers everything from asynchronous processing to zip files
- Plays well with others
 - Easy integration with C / C++, VM's available for JAVA
- Runs everywhere
 - Cross platform and portable
- Open source
 - The Python implementation is under an open source license that makes it freely usable and distributable, even for commercial use
- Strong community

Quality Community: Python Enhancement Proposals

- <http://www.python.org/dev/peps/>
- Well thought on rationale for changes / enhancements / philosophy
- Cogent motivation, debate, and relative merits
- Great place to learn development paradigms in general

- ABOUT >>
- NEWS >>
- DOCUMENTATION >>
- DOWNLOAD >>
- COMMUNITY >>
- FOUNDATION >>
- CORE DEVELOPMENT >>**

[Why Develop Python?](#)[Getting Set Up](#)[Issue Workflow](#)[How to Contribute to Python](#)[Intro to Development](#)[Development Process](#)[Culture](#)[Tools](#)[Patch Submission](#)[Buildbot](#)[Documenting Python](#)[FAQ](#)**PEP Index**[python-dev Summaries](#)[Python.org](#)[Browse Subversion](#)[Daily Snapshots](#)[LINKS >>](#)**Help Fund Python****PEP:** 8**Title:** Style Guide for Python Code**Version:** 68852**Last-Modified:** [2009-01-22 09:36:39 +0100 \(Thu, 22 Jan 2009\)](#)**Author:** Guido van Rossum <guido@python.org>, Barry Warsaw <barry@python.org>**Status:** Active**Type:** Process**Created:** 05-Jul-2001**Post-History:** 05-Jul-2001

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python[1].

This document was adapted from Guido's original Python Style Guide essay[2], with some additions from Barry's style guide[5]. Where there's conflict, Guido's style rules for the purposes of this PEP. This PEP may still be incomplete (in fact, it may never be finished <wink>).

A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 [6] says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

Is Python a scripting language?

- Historically scripting refers to authoring simple tasks, generally in a high level interpreted language. Today the meaning is less clear.
- Scripting probably better refers to Python's approach to development, rather than what is capable in the language
 - Scripting fosters an exploratory, incremental approach to programming
 - The developer scales up in complexity and power as necessary

Common use cases

- Python code is often deployed in the context of larger applications
 - Integration with Labview, GIS, Maya, Sage
- Coordinating heterogeneous software components
 - linking software written in diverse languages
- Rapid prototyping
 - exploring ideas before a detailed implementation
- Concatenative programming
 - creating software by intertwining libraries

Rethinking performance

- Key notion: developer vs. computational efficiency
 - common misconception that code is often CPU bound
 - direct your time where it's valuable
- Like JAVA, Python is compiled to byte-code
 - Portability at the expense of speed
 - The core language, however, is highly optimized
 - built-in data types are implemented in C
 - built-in methods are thoughtful - sort is approximately 1200 lines of C in later versions of Python
- More commonly than JAVA, you'll see Python deployed in high performance environments - Boost / LLNL

Getting started:
Language fundamentals, core data
types, control flow

Python versions

- Python 3000 (released early 2009)
- Intentionally backwards incompatible
 - Major changes:
 - print is a function (previously a statement), API modifications (often views and iterators instead of lists), text vs. data instead of unicode vs. 8-bit
 - 2to3 tool available
- Reference:
<http://docs.python.org/3.0/whatsnew/3.0.html>

Sample Python program (3.0)

```
Terminal — emacs — 59x27
import math

# x: circumference of the circumscribed (outside) polygon
# y: circumference of the inscribed (inside) polygon

# max error allowed
eps = 1e-10

# initialize w/ square
x = 4
y = 2*math.sqrt(2)

ctr = 0
while x-y > eps:
    xnew = 2*x*y/(x+y)
    y = math.sqrt(xnew*y)
    x = xnew
    ctr += 1

print("PI = " + str((x+y)/2))
print("# of iterations = " + str(ctr))

-uu-:---F1 hello_world.py All L9 (Python)-----
Wrote /Users/josh/temp/hello_world.py
```

Sample program (2.x)

```
Terminal — emacs-i386 — 70x14
def speak():
    foo = 'We are the knights who say ...'
    options = ['charge?', 'banana?', 'neet!']
    for option in options:
        result = foo.replace('...', option)
        print result
    if 'neet' in result:
        print "\tThat's more like it!"

speak()

--uu-:----F1 knights.py All L1 (Python)-----
```

```
Terminal — bash — 65x6
Josh:temp josh$ python knights.py
We are the knights who say charge?
We are the knights who say banana?
We are the knights who say neet!
    That's more like it!
Josh:temp josh$
```


Tips for learning languages

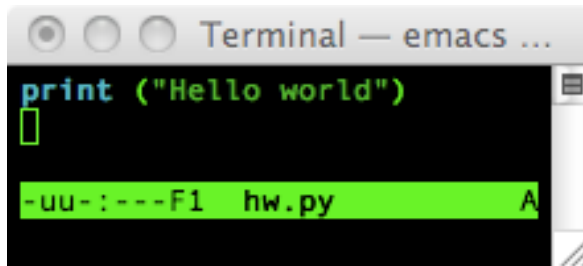
- Strategies that have proven valuable to last semester's students
 - Iteration and refinement - first make it work, then make it elegant
 - Copious examples
 - Side by side comparison against a familiar language
 - Think before using the web as a crutch

Writing and running programs

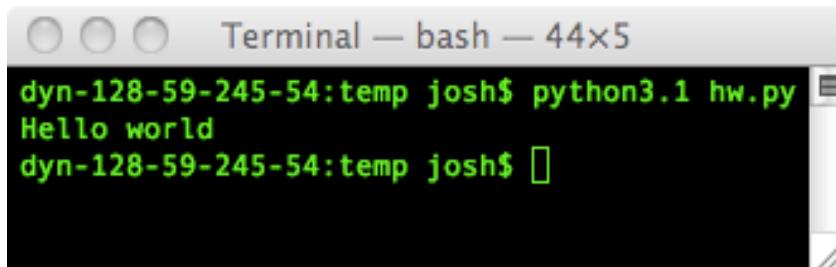
- Python programs are run by passing them to the interpreter (or via an interactive session)
- Code has the extension (.py) - compiled bytecode (.pyc)
- Python interpreters compile statements to byte code and execute them on a virtual machine
- Compilation occurs automatically the first time the interpreter reads code which imports modules, and subsequently upon modification
 - Standalone scripts are recompiled when run directly
- Compilation and execution are transparent to the user
- Python includes an interactive session mode

Hello world

Via .py files and the console

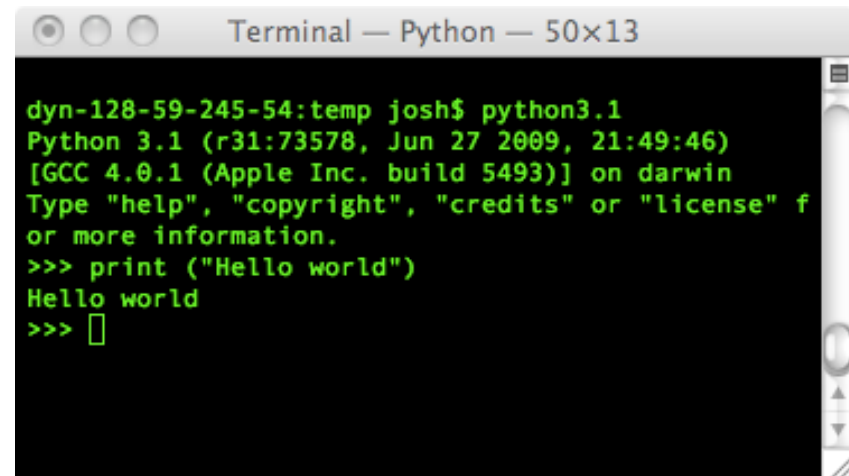


```
Terminal — emacs ...  
print ("Hello world")  
█  
-uu-:---F1 hw.py A
```



```
Terminal — bash — 44x5  
dyn-128-59-245-54:temp josh$ python3.1 hw.py  
Hello world  
dyn-128-59-245-54:temp josh$ █
```

Via the interpreter



```
Terminal — Python — 50x13  
dyn-128-59-245-54:temp josh$ python3.1  
Python 3.1 (r31:73578, Jun 27 2009, 21:49:46)  
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin  
Type "help", "copyright", "credits" or "license" f  
or more information.  
>>> print ("Hello world")  
Hello world  
>>> █
```

Main methods

```
Terminal — emacs — 77x24
print ("Code in out scope is executed")
print ("as the module loads")

def foo():
    # classes and functions are declared and parsed
    # but not executed unless called
    ....
    ....

def main():
    # a good place to parse command line arguments
    # or place testing code for the module

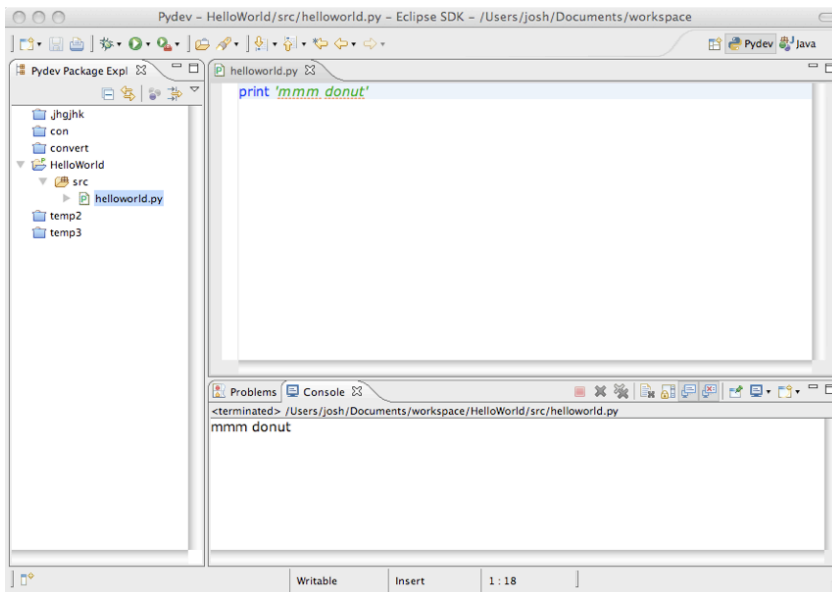
# a common pattern to check whether this is the main module
# makes use of the built-in dictionary
if __name__ == "__main__":
    print ("I'm the main module!")
    main() # calls main

# if this wasn't the first module loaded, return control here

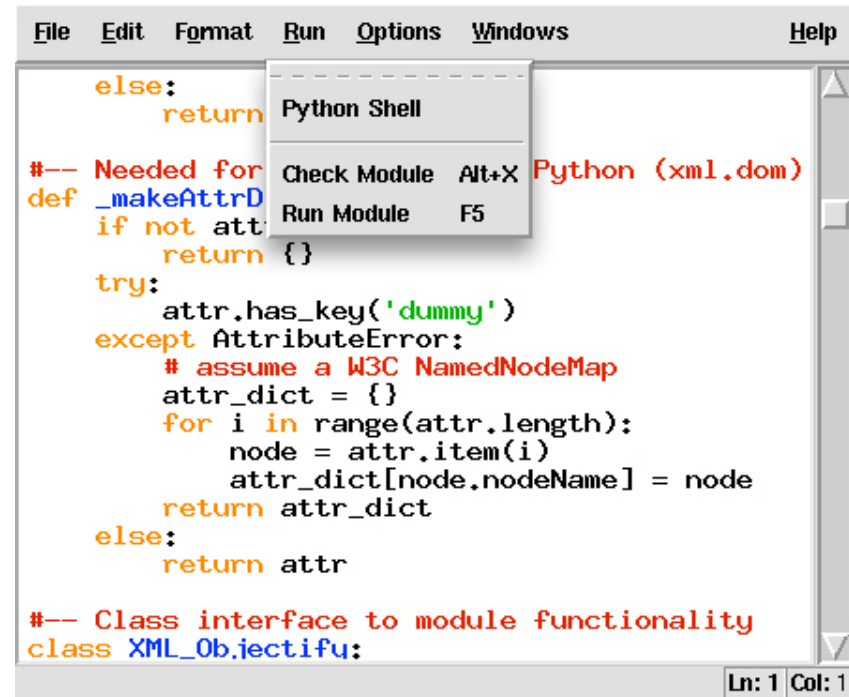
-uu-:---F1 hw.py All L12 (Python)-----
Wrote /Users/josh/temp/hw.py
```

Development Environments: Text Editors vs. IDEs vs. IDLE

Eclipse with PyDev



IDLE



<http://pydev.sourceforge.net/>

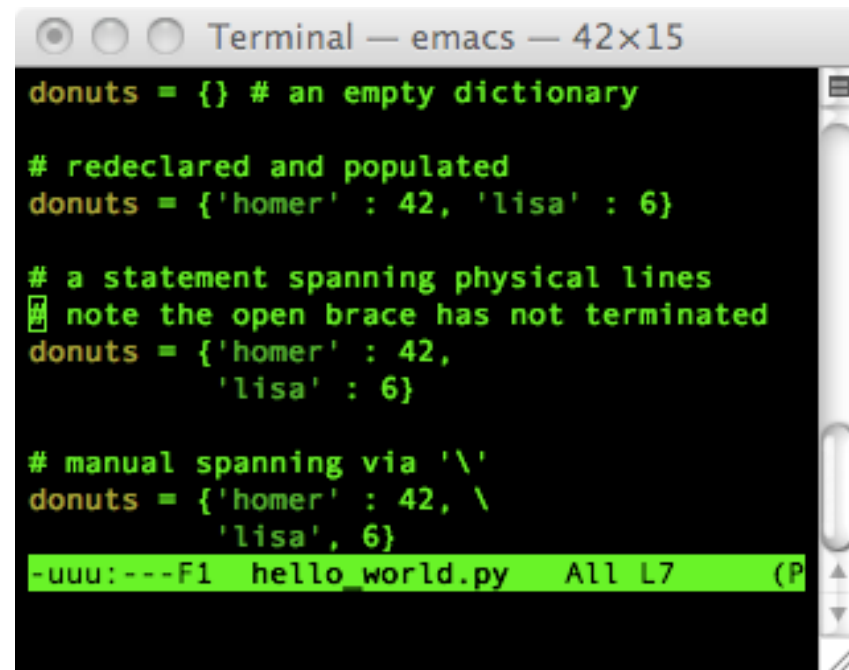
Lexical Structure:

Python cares about whitespace

- The lowest level syntax of the language - specifies, for instance, how comments and variable names appear
- Python programs are composed of a set of plain text ASCII encoded source files
- Like other language, Python source files are a sequence of characters, however: unlike in C or JAVA, in Python we're interested in lines and indentation - whitespace counts

Logical vs. Physical Lines

- Python programs consist of a sequence of logical lines, which may contain one or more physical lines
 - lines may end with a comment
 - blank lines are ignored by the compiler
 - the end of a physical line marks the end of a statement
- Producing readable code:
 - physical lines may be joined by a (`\`) character
 - if an open parenthesis (`()`), brace (`{}`), or bracket (`[]`) has not yet been closed, lines are joined by the compiler



```
Terminal — emacs — 42x15
donuts = {} # an empty dictionary

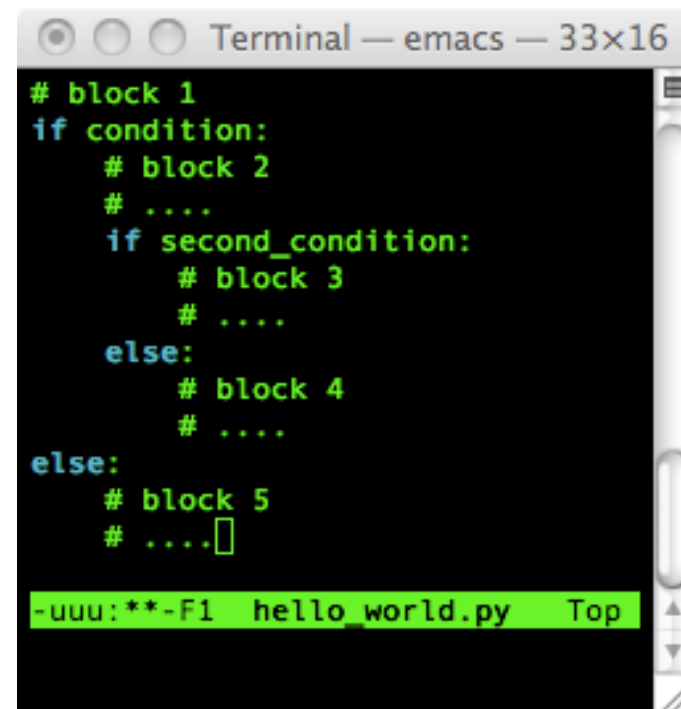
# redeclared and populated
donuts = {'homer' : 42, 'lisa' : 6}

# a statement spanning physical lines
note the open brace has not terminated
donuts = {'homer' : 42,
          'lisa' : 6}

# manual spanning via '\ '
donuts = {'homer' : 42, \
          'lisa', 6}
-uuu:--F1 hello_world.py All L7 (P
```

Indentation

- Indentation is used to express block structure
- Unlike C or JAVA (or most languages in fact) indentation is the only way to denote blocks
- Blocks are delineated by contiguous whitespace sequence, typically in units of four spaces, immediately to the left of a statement
- All statements indented by the same amount belong to an identical block
- Indentation applies only to the first physical line in a logical block
- The first physical line in a source file must have no indentation



```
Terminal — emacs — 33x16
# block 1
if condition:
    # block 2
    # ....
    if second_condition:
        # block 3
        # ....
    else:
        # block 4
        # ....
else:
    # block 5
    # ....
-uuu:**-F1 hello_world.py Top
```


Formal syntax: Tokens, identifiers, keywords, operators, delimiters

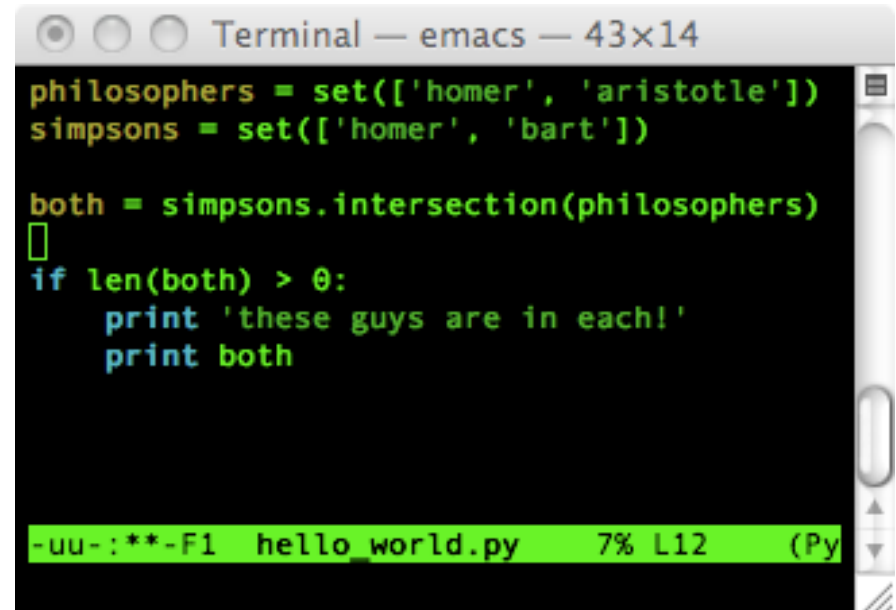
- Logical lines are understood as sequences of tokens
- Tokens are substrings of the line, which correspond to identifiers, keywords, operators, delimiters, and literals
- Identifiers name functions, variables, classes, or modules
 - Identifiers start with a character or an underscore
 - Python is case sensitive
 - Punctuation is disallowed within identifiers
 - Convention: start a class name with an uppercase character, and everything else with a lower
- A literal is a string or numerical value that appears directly in program text
- Keywords are reserved identifiers - Python has about 30 of them, many should be familiar to other languages
 - (and, assert, break, class, continue, def, del, and so forth)
- Non-alphanumeric characters (and combinations) are used by Python as operators
 - (+, -, *, /, <=, <>, !=, and so forth)
- These symbols are used as delimiters in expressions, lists, dictionaries, and sets
 - ((,), [,], {, }, , and so forth)

Statements and Expressions

- A Python program may be understood as a sequence of simple and compound statements
- Unlike C or JAVA, Python does not have any forward declarations or other top level syntax
- The general rule is one statement per line (statements can be terminated with (;), but it's unusual style)
- Statements may be expressions (a phrase which evaluates to produce a value) or assignments
- The simplest expressions are literals and identifiers
- Expressions are built by joining subexpressions with operators and identifiers

Compound statements

- Contain a set of statements and control their execution
- Compound statements contain a set of clauses
 - Each clause has a header starting with a keyword, and ending with a (:)
 - Followed by a body, which is itself a sequence of statements, terminated when indentation returns to the outer level
- Also legal are simple statements following the (:)

A terminal window titled "Terminal — emacs — 43x14" with a black background and green text. The code defines two sets: 'philosophers' containing 'homer' and 'aristotle', and 'simpsons' containing 'homer' and 'bart'. It then calculates the intersection of these two sets into a variable 'both'. An if statement checks if the length of 'both' is greater than 0, and if so, prints the message 'these guys are in each!' and the contents of 'both'. The status bar at the bottom shows '-uu-:**-F1 hello_world.py 7% L12 (Py'.

```
philosophers = set(['homer', 'aristotle'])
simpsons = set(['homer', 'bart'])

both = simpsons.intersection(philosophers)
[]
if len(both) > 0:
    print 'these guys are in each!'
    print both

-uu-:**-F1 hello_world.py 7% L12 (Py
```

Assignment



Python

- Python is dynamically typed and features automatic memory management

```
foo = 42
```

```
foo = 42.42e7
```

```
foo = "dolphin"
```

```
foo = ["one", "two", "three"]
```

```
foo = {'dozen' : 12}
```

```
foo.append("four")
```

```
foo = some_class()
```

C

```
int foo = 42;
```

```
float foo = 42.42;
```

```
char *cp = "dolphin";
```

Java

```
List<String> foo = new  
ArrayList<String>();
```

```
Object foo = new Object ();
```

Data types

Built-in types

- All data values in Python are objects, and each object has a type
- Built-in types cover numbers, strings, lists, tuples, and dictionaries
- Type determines supported operations
 - For instances, lists support `.reverse()`, but strings (as immutable objects) do not
- Mutable vs. immutable objects
- Useful functions: `type(obj)` and `isinstance(obj, type)`
- User defined type are supported via classes

Basic types: Numbers

- Support for ints (including long, if you're coming from C) floating point, and complex
- Unlimited precision (you may indeed compute $2^{1000,000}$ (but not if you're in a rush))
- Up casting is automatic
- All numbers in Python are immutable - so any operation on a number always produces a new object

Syntax	Result
4, -24, 0	Standard Integer (corresponds to C longs)
12.4, 3.14e-10, 4.0e+210	Floating Point (corresponds to C doubles)
1e10, 1E1	Scientific Notation
9L	Unlimited Precision Long Integer
3+4j	Complex
0177, 0x9ff, 0xFF	Octal and hex literals for integers

Basic types: Numeric operations

- Basic mathematical modules are included by default
 - You'll need "import math" for square root, common constants, etc
- NumPy / SciPy: <http://numpy.scipy.org/>
 - Similar to a high performance open source MATLAB implementation
 - Matrix data types, vector processing, sophisticated computation libraries
- Sage: www.sagemath.org

Syntax	Result
1 + 1	2
1 - 2.0	-1.0
2 * 12	24
5 / 2	2
5.0 / 2	2.5
5 % 2	1
2 ** 8.1	274.37
2 * 5e100	9.99e+100
2.0 * 10e1	200.0
math.pi	3.141592...
math.sqrt(85)	9.2195...

Basic types: Strings

- Strings are immutable objects which store a sequence of characters (plain or Unicode)
 - May be used to represent arbitrary sequences of binary bytes
- Single or double quoting allowed, triple quoting for multiline
- Unicode support
 - Useful for multilingual text and special characters
 - Very strong support in Python 3000
- Raw string encoding
 - `r"C:\Josh\Document's\"`
- All the typical escape characters
 - `"\n"` – new line
 - `"\t"` – tab

Syntax	Result
<code>'Marge'</code>	<code>'Marge'</code>
<code>"Homer"</code>	<code>'Homer'</code>
<code>"Lisa's Music"</code>	<code>"Lisa's Music"</code>
<code>"Homer's\tdonut"</code>	<code>"Homer's donut"</code>
<code>u'bart\u0026lisa'</code>	<code>u'bart&lisa'</code>
<code>r"C:\Simpson\Bart\"</code>	<code>"C:\\Simpson\\Bart"</code>

Basic types: String methods

Syntax	Result
<code>s1 = "Josh"</code> <code>s2 = "a marathon"</code>	Declares a new variable
<code>s1 + " is training for " + s2</code>	'Josh is training for a marathon' (concatenation)
<code>s1[0]</code>	'J' (give me the first character in s1)
<code>s1[0:3]</code>	Jos (give me the first 3 characters)
<code>s1 * 3</code>	"JoshJoshJosh" (repeat)

Syntax	Result
<code>foo = "The Simpsons"</code>	Declares a new variable
<code>foo.lower()</code>	'the simpsons'
<code>foo.find("S")</code>	4
<code>foo.find("z")</code>	-1
<code>foo.split(" ")</code>	["The", "Simpsons"]
<code>foo.islower()</code>	False
<code>"The" in foo</code>	True
<code>" whitespace ".strip()</code>	"whitespace"

Common theme with Python: basic operations are abundant and behave as you would expect - the complexity of syntax corresponds with the complexity of the operation

Basic types: Lists

- A list is an ordered collection of objects
- Lists are both mutable and heterogeneous (may be composed of arbitrary objects of different types)
- Support arbitrary nesting (e.g., lists within lists)
- Support slicing and indexing (by offset)

<code>foo = []</code>	(an empty list)
<code>foo = [0,1,"bar"]</code>	(a list containing two integers and a string)
<code>foo[2]</code>	"bar" (the second element in foo)
<code>foo = [0,1,["a", "nested", "list"], 3]</code>	(a nested list)
<code>foo[2]</code>	["a", "nested", "list"] (the third element of Foo)
<code>foo[2][1]</code>	"nested" (the second element of the third element of foo)

Basic types: List methods

- Modifying the list (in any function) modifies the original copy)
- Sorting is a built-in method (common feature of scripting languages)
 - Highly optimized, handles many special common cases (including partially sorted list, a list constructed from two sorted lists, a reversed list, etc)
- Basic slicing and indexing:
 - `foo = ["a", "b", "c", "d"]`
 - `foo[-2]`
 - Counts from the right, returns 'c'

<code>foo = []</code>	(empty list)
<code>foo.append("a")</code>	<code>["a"]</code>
<code>foo.append("c")</code>	<code>["a", "c"]</code>
<code>foo.append("b")</code>	<code>["a", "c", "b"]</code>
<code>foo.append("x")</code>	<code>["a", "c", "b", "x"]</code>
<code>foo.pop()</code>	"x" (removes and returns the last element from Foo)
<code>foo.sort()</code>	<code>["a", "b", "c"]</code> (sorts in place)
<code>foo.index("c")</code>	2
<code>foo[1:]</code>	<code>["b", "c"]</code> (slicing: give me all the elements beginning from index 1)

Ex. Iterating over a list

```
Terminal — emacs-i386 — 70x14
def speak():
    foo = 'We are the knights who say ...'
    options = ['charge?', 'banana?', 'neet!']
    for option in options:
        result = foo.replace('...', option)
        print result
    if 'neet' in result:
        print "\tThat's more like it!"

speak()

--uu-:---F1 knights.py All L1 (Python)-----
```

```
Terminal — bash — 65x6
Josh:temp josh$ python knights.py
We are the knights who say charge?
We are the knights who say banana?
We are the knights who say neet!
    That's more like it!
Josh:temp josh$
```

Nested lists ex. (but NOT the right way to represent a matrix)

```
def main():  
    matrix = ["upper left",2,3],[4,"center",6],[7,8,"lower right"]  
    print matrix[1][1]  
    print "The upper row" + matrix[0]
```

main()

→

center

The upper row: ['upper left', 2, 3]

Basic Types: Dictionaries

- Dictionaries are unordered collections
- Like lists, dictionaries are mutable and heterogeneous
- Unlike lists, which are accessed by index, dictionaries are accessed by **key**
- Methods `.keys()`, `.values()`, `.items()` vs. `iterkeys()`, etc
- Dictionaries are mappings: arbitrary collections of objects indexed by (almost) arbitrary keys
- One of the more optimized types in Python, used properly many operations are constant time (amortized)

Syntax	Result
<code>foo = {}</code>	Empty dictionary
<code>foo['coffee'] = "good"</code>	Single item dictionary
<code>foo['decaf'] = "bad"</code>	Foo now contains two items
<code>foo['coffee']</code>	"good"
<code>'decaf' in foo</code>	True
<code>foo.keys()</code>	['coffee', 'decaf']
<code>foo['tea']</code>	KeyError!
<code>foo['tasty'] = ['cookies', 'ice-cream']</code>	Note that dictionary keys may reference arbitrary objects

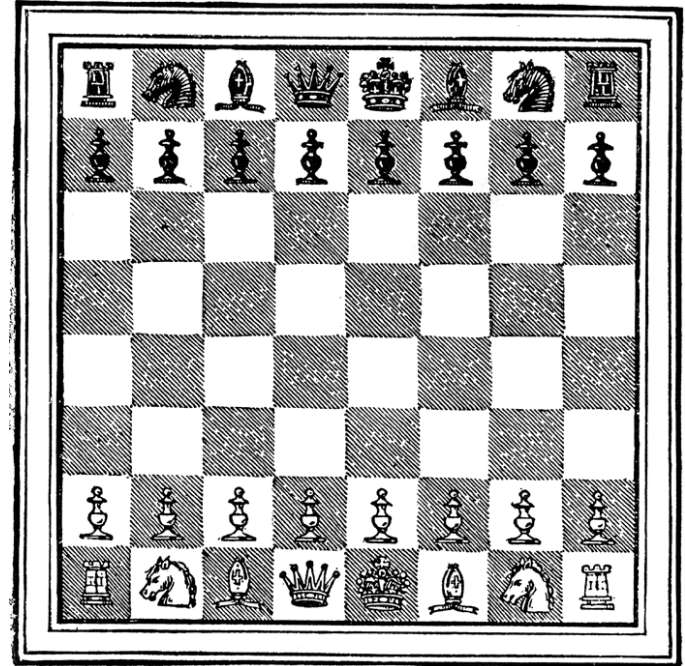
Discussion:

Why can you not sort a dictionary in place?

What would you do if you had to sort one?

Dictionary basics: sparse data structures

- `def main():`
- `board = {}`
- `# a chess board`
- `# say we only want to keep track of the`
- `# position of the kings`
- `board[(0, 4)] = "LIGHT_KING"`
- `board[(7, 4)] = "DARK_KING"`



Discussion:

- Can you compare the memory usage of this representation against a matrix using lists? (Chess boards are 8x8)
- Ignore the size of the dictionary and list support code (becomes insignificant at large sizes)

Dictionaries as record objects

```
import random
homer['donutSupply'] = 5
homer['hairsRemaining'] = 4
homer['noises'] = ['Munch', 'Crunch', 'MMM Donut', 'Aghggh']
while homer['donutSupply'] > 0:
    homer['donutSupply'] -= 1
    print random.choice(homer['noises'])
```

→

```
MMM Donut
Aghggh
Aghggh
Munch
Crunch
```

Basic types: Tuples

- Tuples are:
 - Ordered collections
 - Accessed by offset
 - Static and immutable
 - Fixed length
 - Heterogeneous
 - Nestable
- Many of the same operations as lists and dictionaries
- Why tuples?
 - Program integrity
 - Occasionally used as a type of constant

Syntax	Result
<code>()</code>	An empty tuple
<code>foo = ('homer', 'marge', 42)</code>	A three item tuple
<code>foo[1]</code>	<code>'Marge'</code>
<code>len(foo)</code>	3
<code>foo[-1]</code>	42
<code>foo[:-1]</code>	<code>['homer', 'marge']</code> (give me everything up until the last element)

Control flow basics

Comparisons and Booleans

- Parenthesis are optional
- The end of **line** is the end of **statement**
- The end of **indentation** is the end of a **block**
- Why indentation syntax?
 - Enforces consistency and readability

C++:

```
if (x > y){  
    cout << "x is larger";  
}else {  
    cout << "x is smaller";  
}
```

Python:

```
if x > y:  
    print "x is larger"  
else:  
    print "x is smaller"
```

Comparisons and Booleans Continued

C, C++, Java:

```
if (x)
    if (y)
        statement1;
else
    statement2;
```

- The dangling else problem:
 - Which statement does the else belong to?
- This problem doesn't occur in Python
- Note: although you may be smart enough not to write code this way, others on your project may not be

Comparisons and Booleans Continued

Syntax	Result
foo = True	A new boolean variable
5 > 5	False
5 == 5, 5 == 4	(True, False)
not 5	False
'x' == "x"	True
(True and True and False)	False
True and (5 > 4)	True
True and ("a" == "b")	False
True or False	True

- Python:
- If x:
 - if y:
 - statement1
 - else:
 - statement2

Control flow basics

Control flow: for statements

For loops are valid over any iterable (strings, lists, tuples)

```
for x in range(5):  
    print x
```

->

0

1

2

3

4

```
Simpsons = ["Homer", "Marge",  
            "Lisa", "Bart"]
```

```
for person in Simpsons:  
    print person
```

->

Homer

Marge

Lisa

Bart

Control flow: nested for statements

- Loops (like all statements) can be nested
- For loops are valid over any iterable sequence

```
i = 0
for x in range(3):
    for y in range(3):
        i = i + 1
        #or i += 1
print i
```

Control flow: while statements

- General form:
- While <test>: #repeat while true
 - <statements>
- Else: #optional
 - <statements>

```
a = 0
b = 10
while a < b:
    a += 1
    print a
```

Control flow: break and continue

- **break:**
 - exits the loop immediately
- **continue:**
 - returns to the beginning of the loop

```
a = 0
b = 100
while a < b:
    a += 1
    if a % 2 == 0:
        continue
    b = b / 2
    print a
    if b < 10:
        break
```

Preview: list comprehension

- Likely novel if you're familiar with C or Java
 - Derived from set theory
- A shortcut to construct a new list
 - Also a performance boost due to Python's internals
- We'll see these used more later

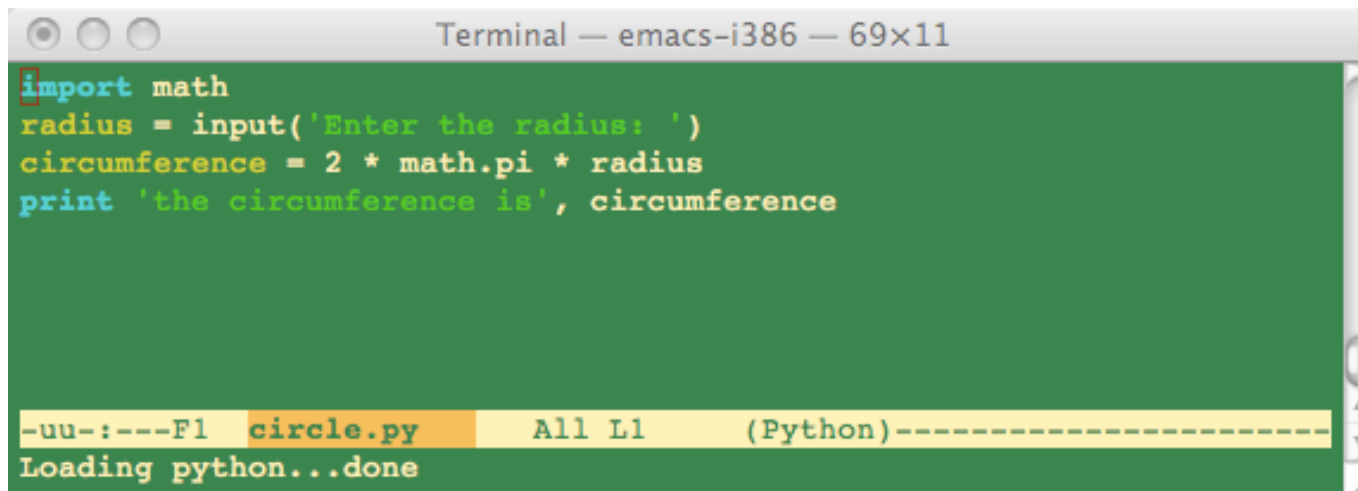
```
foo = [1, 2, 3]
bar = [x + 10 for x in foo]
→
bar is now [11, 12, 13]
```

#Identical to:

```
bar = []
for x in foo:
    bar.append(x + 10)
```

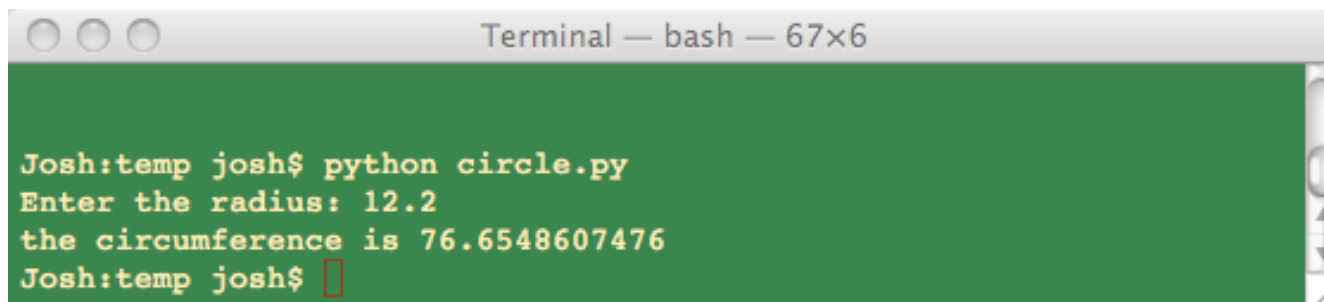
User input from the console

Reading from the console



```
Terminal — emacs-i386 — 69x11
import math
radius = input('Enter the radius: ')
circumference = 2 * math.pi * radius
print 'the circumference is', circumference

-uu-:---F1 circle.py All L1 (Python)-----
Loading python...done
```



```
Terminal — bash — 67x6

Josh:temp josh$ python circle.py
Enter the radius: 12.2
the circumference is 76.6548607476
Josh:temp josh$
```

Type casting

- What happens if you try to square a number the user enters?
- Correction
 - `print int(userSays) ** 2`
- Casts work the other way too
 - `x = "hi" + 5 + " times"`
 - `x = "hi" + str(5) + " times"`
- You may cast between complicated structures as well – but be careful you understand the expected behavior in advance

```
while True:
x = input('Square: ')
if (x == 'quit'):
    break
else:
    print x ** 2

→
5
TypeError: unsupported operand for 'str'
```

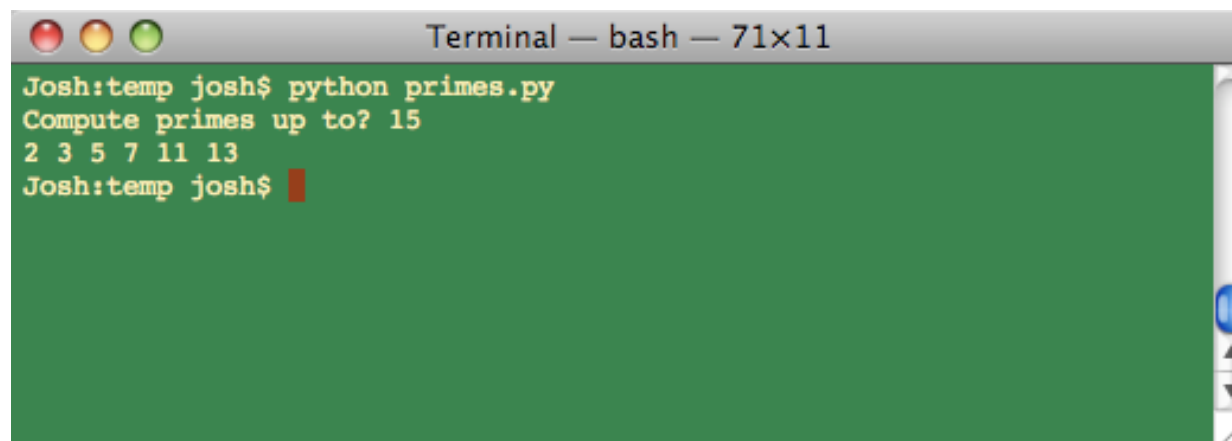
Assignment 1

Due before the start of class next
week

Exercise 1 of 2

Primes

- 1. Install Python
- 2. Write a script to compute and print the prime numbers below N. Read N from the console by prompting the user.
 - Performance is not important, you'll be graded on correctness only.



```
Terminal — bash — 71x11
Josh:temp josh$ python primes.py
Compute primes up to? 15
2 3 5 7 11 13
Josh:temp josh$
```

Exercise 2 of 2

Word frequency

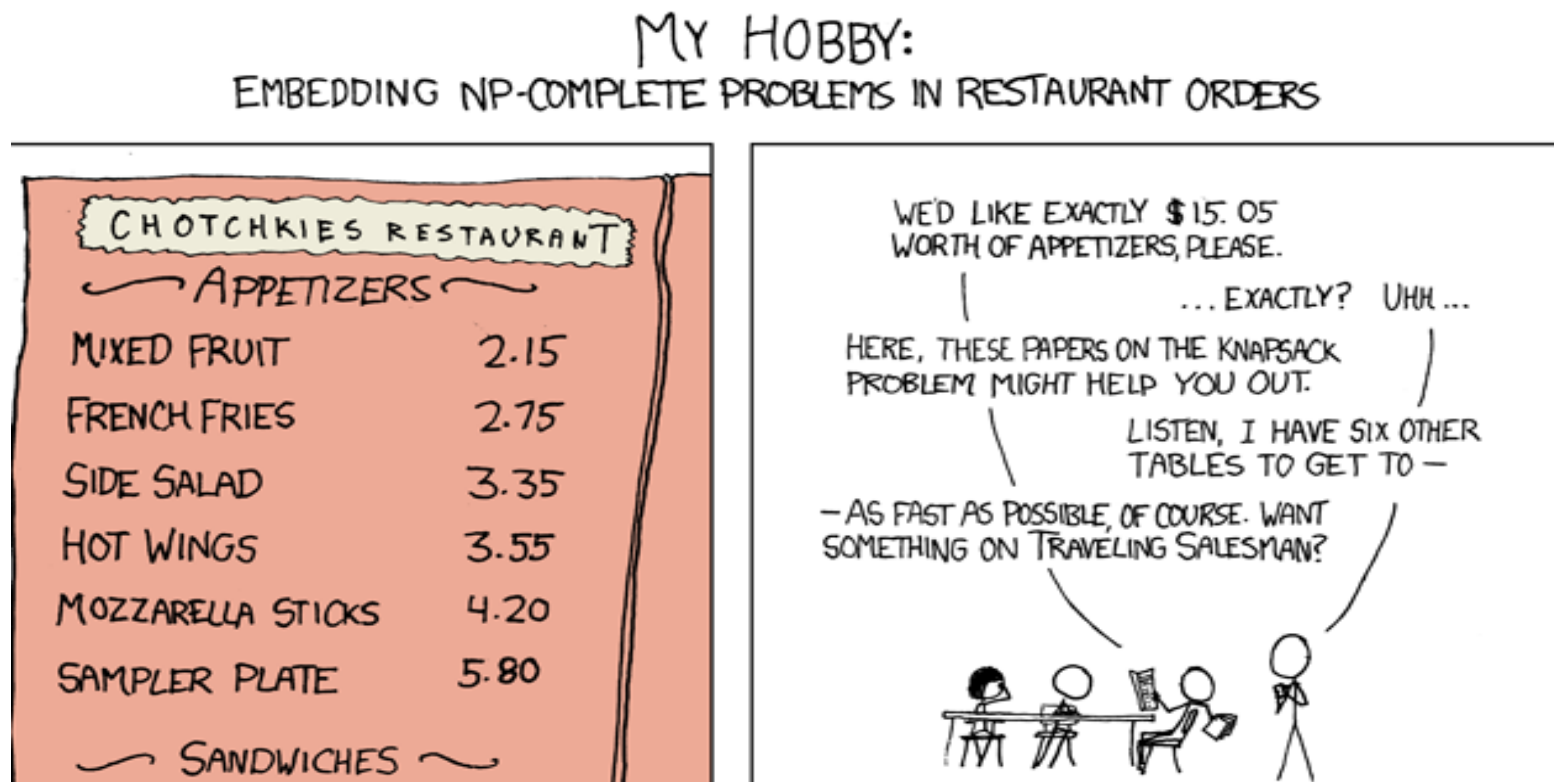
- Write a word frequency counter. Your program should prompt the user to enter a single word per line (or return to finish). Report the frequency of each word, sorted alphabetically in ascending order.
- Hint: Use a dictionary. When the user enters a word, check if the dictionary has that key.

```
$python word_freq.py
Enter a word (or return to finish): homer
Enter a word (or return to finish): homer
Enter a word (or return to finish): homer
Enter a word (or return to finish): bart
Enter a word (or return to finish):
```

```
You entered:
bart : 1
homer: 3
```

Challenge problem

- For the ambitious only, implement a solution to the following comic (a variant of the knapsack problem).



Source <http://xkcd.com/287>

Submission instructions

- Please submit a zip including your source and README to courseworks
- Under class files, select “Post File” – be sure to select the appropriate folder (e.g., assignment 1)
- CVN: email me a zip (we’ve had issues uploading code)
- Note:
 - When you’re reading files down the road, use relative (not absolute paths)

See you next week

- Course website (with these slides and the assignment)
 - www.cs.columbia.edu/~joshua/teaching/cs3101/
- Bring your laptops next week
- Questions: joshua@cs.columbia.edu