Python Crash Course For Developers

Ephraim Berkovitch

Overview

- History
- Installing & Running Python
- Names & Assignment
- Sequences types: Lists, Tuples, and Strings
- Mutability

History

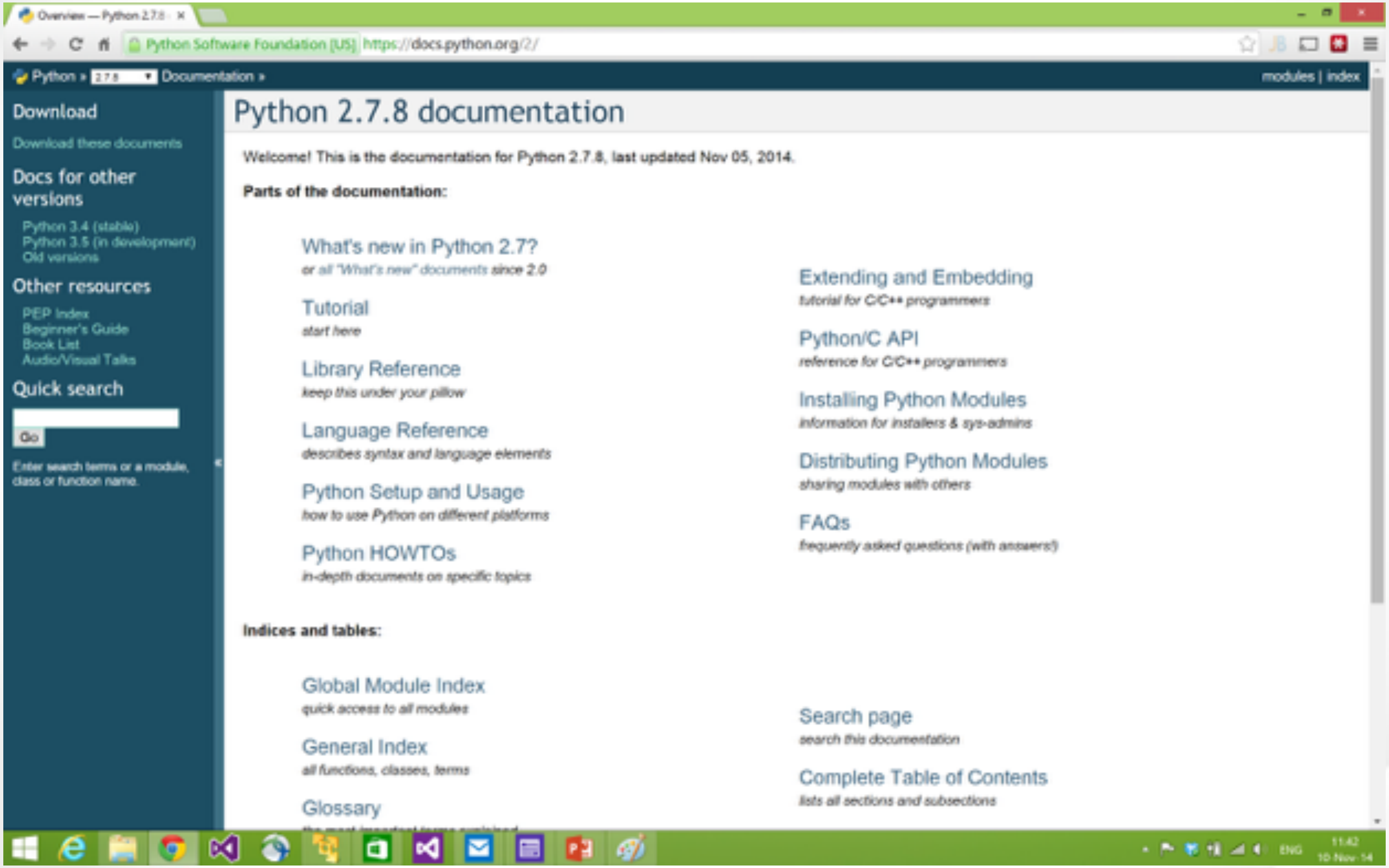
- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning
- Considered a scripting language, but is much more
- Scalable, object oriented and functional from the beginning
- Used by Google from the beginning
- Increasingly popular

Ideology

- “Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressive-ness is endangered.”

  - Guido van Rossum

http://docs.python.org



Overview — Python 2.7.8

Python Software Foundation [US] https://docs.python.org/2/

Python » 2.7.8 Documentation »

modules | index

Download

Download these documents

Docs for other versions

- Python 3.4 (stable)
- Python 3.5 (in development)
- Old versions

Other resources

- PEP Index
- Beginner's Guide
- Book List
- Audio/Visual Talks

Quick search

Go

Enter search terms or a module, class or function name.

Python 2.7.8 documentation

Welcome! This is the documentation for Python 2.7.8, last updated Nov 05, 2014.

Parts of the documentation:

What's new in Python 2.7?
or all "What's new" documents since 2.0

Tutorial
start here

Library Reference
keep this under your pillow

Language Reference
describes syntax and language elements

Python Setup and Usage
how to use Python on different platforms

Python HOWTOs
in-depth documents on specific topics

Extending and Embedding
tutorial for C/C++ programmers

Python/C API
reference for C/C++ programmers

Installing Python Modules
information for installers & sys-admins

Distributing Python Modules
sharing modules with others

FAQs
frequently asked questions (with answers!)

Indices and tables:

Global Module Index
quick access to all modules

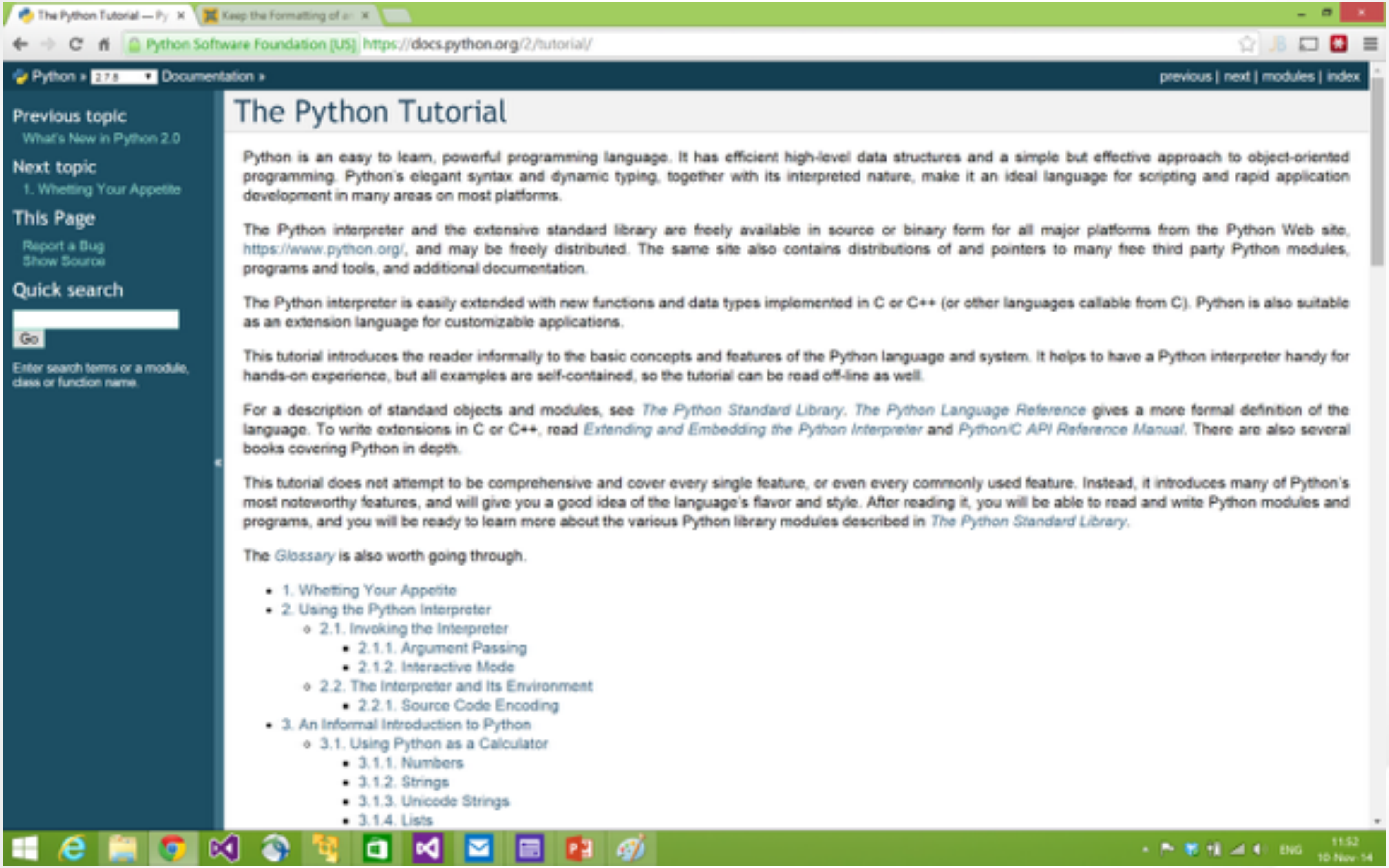
General Index
all functions, classes, terms

Glossary

Search page
search this documentation

Complete Table of Contents
lists all sections and subsections

Python Tutorial


The Python Tutorial — Py
Keep the Formatting of a
Python Software Foundation [US] https://docs.python.org/2/tutorial/
Python » 2.7.8 Documentation »
previous | next | modules | index
Go
ENG
11:52
10-Nov-14

Previous topic
What's New in Python 2.0

Next topic
1. Whetting Your Appetite

This Page
Report a Bug
Show Source

Quick search

Enter search terms or a module, class or function name.

The Python Tutorial

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

For a description of standard objects and modules, see *The Python Standard Library*. *The Python Language Reference* gives a more formal definition of the language. To write extensions in C or C++, read *Extending and Embedding the Python Interpreter* and *Python/C API Reference Manual*. There are also several books covering Python in depth.

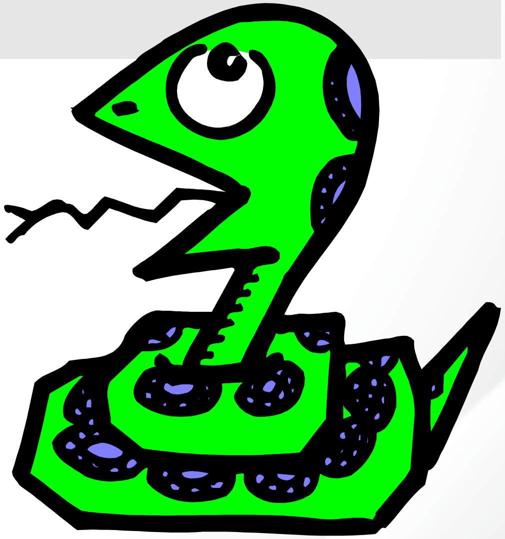
This tutorial does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python's most noteworthy features, and will give you a good idea of the language's flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in *The Python Standard Library*.

The *Glossary* is also worth going through.

Releases

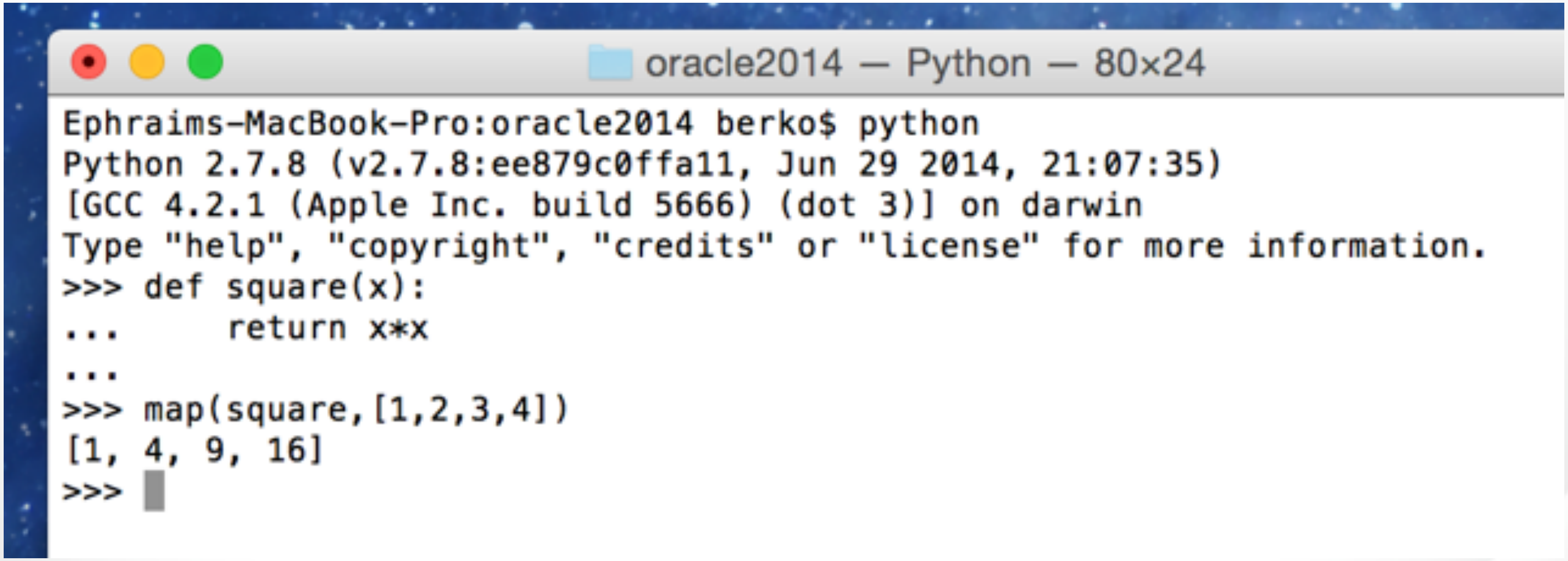
- Created in 1989 by Guido Van Rossum
- Python 1.0 released in 1994
- Python 2.0 released in 2000
- Python 3.0 released in 2008
- Python 2.7 is the recommended version
- 3.0 adoption will take a few years

Running Python



The Python Interpreter

- Typical Python implementations offer both an interpreter and compiler
- Interactive interface to Python with a read-eval-print loop



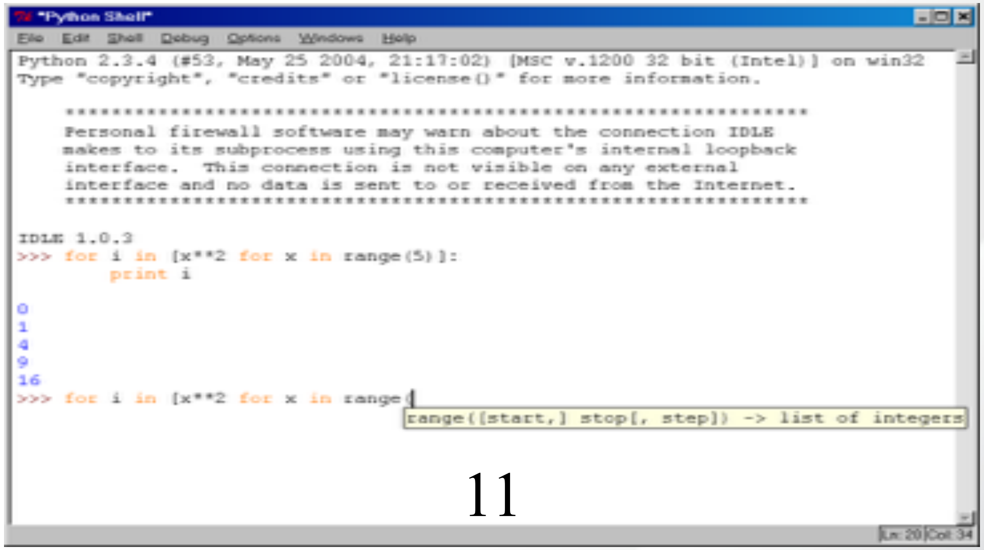
```
Ephraims-MacBook-Pro:oracle2014 berko$ python
Python 2.7.8 (v2.7.8:ee879c0ffa11, Jun 29 2014, 21:07:35)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def square(x):
...     return x*x
...
>>> map(square,[1,2,3,4])
[1, 4, 9, 16]
>>>
```

Installing

- Python is pre-installed on most Unix systems, including Linux and MAC OS X
- The pre-installed version may not be the most recent one (2.6.2 and 3.1.1 as of Sept 09)
- Download from http://python.org/download/
- Python comes with a large library of standard modules
- There are several options for an IDE
  - IDLE – works well with Windows
  - Emacs with python-mode or your favorite text editor
  - Eclipse with Pydev (http://pydev.sourceforge.net/)

IDLE Development Environment

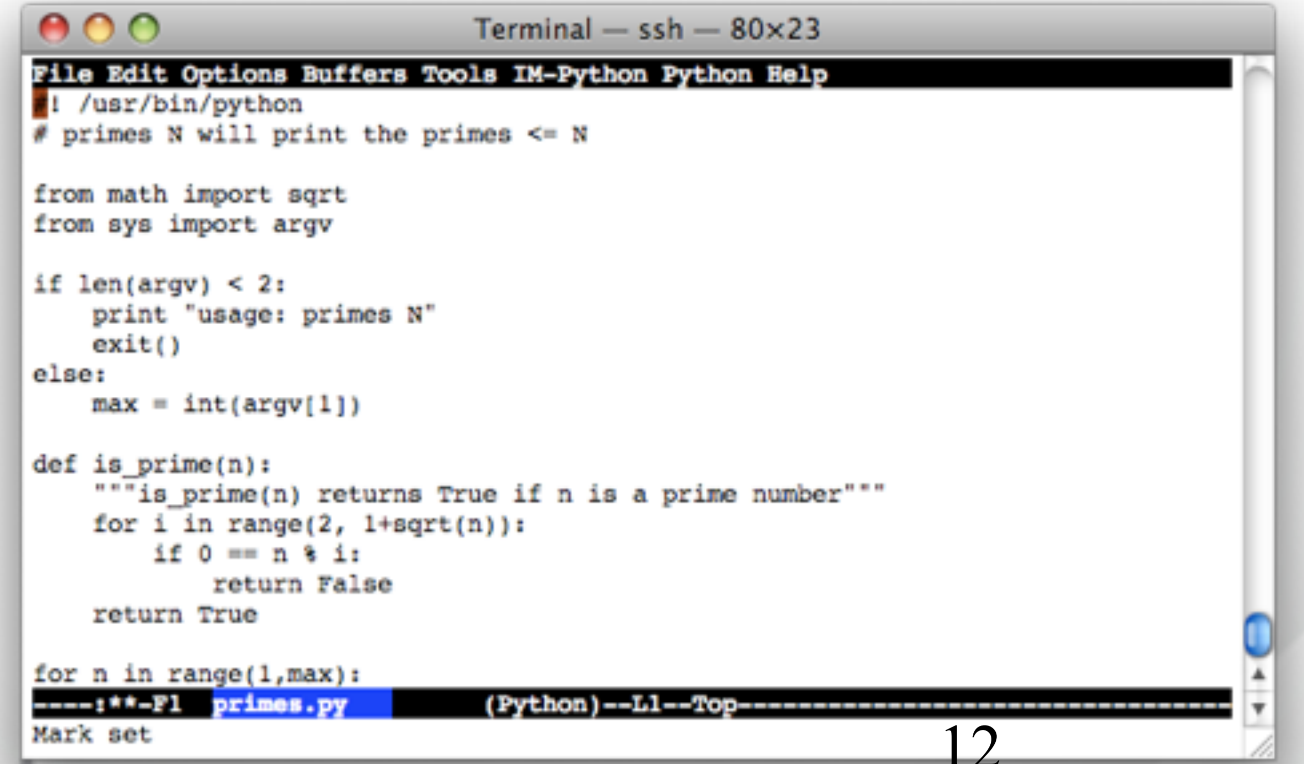
- IDLE is an Integrated DeveLopment Environment for Python, typically used on Windows
- Multi-window text editor with syntax highlighting, auto-completion, smart indent and other.
- Python shell with syntax highlighting.
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility



```
Python 2.3.4 (#53, May 25 2004, 21:17:02) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0.3
>>> for i in [x**2 for x in range(5)]:
        print i

0
1
4
9
16
>>> for i in [x**2 for x in range(
                                   range([start,] stop[, step]) -> list of integers
```

Editing Python in Emacs

- Emacs *python-mode* has good support for editing Python, enabled enabled by default for .py files
- Features: completion, symbol help, eldoc, and inferior interpreter shell, etc.



```
#! /usr/bin/python
# primes N will print the primes <= N

from math import sqrt
from sys import argv

if len(argv) < 2:
    print "usage: primes N"
    exit()
else:
    max = int(argv[1])

def is_prime(n):
    """is_prime(n) returns True if n is a prime number"""
    for i in range(2, 1+sqrt(n)):
        if 0 == n % i:
            return False
    return True

for n in range(1,max):
----:**-F1  primes.py        (Python)--L1--Top-------------------------------
Mark set
```

Running Interactively on UNIX

On Unix…

```
% python
>>> 3+3
6
```

- **Python prompts with ‘>>>’.**
- **To exit Python (not Idle):**
  - In Unix, type CONTROL-D
  - In Windows, type CONTROL-Z + <Enter>
  - Evaluate exit()

Running Programs on UNIX

- Call python program via the python interpreter

  ```
  % python fact.py
  ```

- Make a python file directly executable by
  - Adding the appropriate path to your python interpreter as the first line of your file

    ```
    #!/usr/bin/python
    ```

  - Making the file executable

    ```
    % chmod a+x fact.py
    ```

  - Invoking file from Unix command line

    ```
    % fact.py
    ```

Example ‘script’: fact.py

```
#! /usr/bin/python

def fact(x):
    """Returns the factorial of its argument, assumed to be a posint"""
    if x == 0:
        return 1
    return x * fact(x - 1)

print
print ’N fact(N)’
print "---------"

for n in range(10):
    print n, fact(n)
```

Python Scripts

- When you call a python program from the command line the interpreter evaluates each expression in the file
- Familiar mechanisms are used to provide command line arguments and/or redirect input and output
- Python also has mechanisms to allow a python program to act both as a script and as a module to be imported and used by another python program

Example of a Script

```
#! /usr/bin/python

""" reads text from standard input and outputs any email
    addresses it finds, one to a line.
"""

import re

from sys import stdin

# a regular expression ~ for a valid email address

pat = re.compile(r'[-\w][-.\w]*@[-\w][-\w.]+[a-zA-Z]{2,4}')

for line in stdin.readlines():

    for address in pat.findall(line):

        print address
```

results

```
python> python email0.py <email.txt
bill@msft.com
gates@microsoft.com
steve@apple.com
bill@msft.com
python>
```

Getting a unique, sorted list

```
import re
from sys import stdin

pat = re.compile(r'[-\w][-.\w]*@[-\w][-\w.]+[a-zA-Z]{2,4}’)
# found is an initially empty set (a list w/o duplicates)
found = set( )
for line in stdin.readlines():
    for address in pat.findall(line):
        found.add(address)
# sorted() takes a sequence, returns a sorted list of its elements
for address in sorted(found):
    print address
```

results

```
python> python email2.py <email.txt
bill@msft.com
gates@microsoft.com
steve@apple.com
python>
```

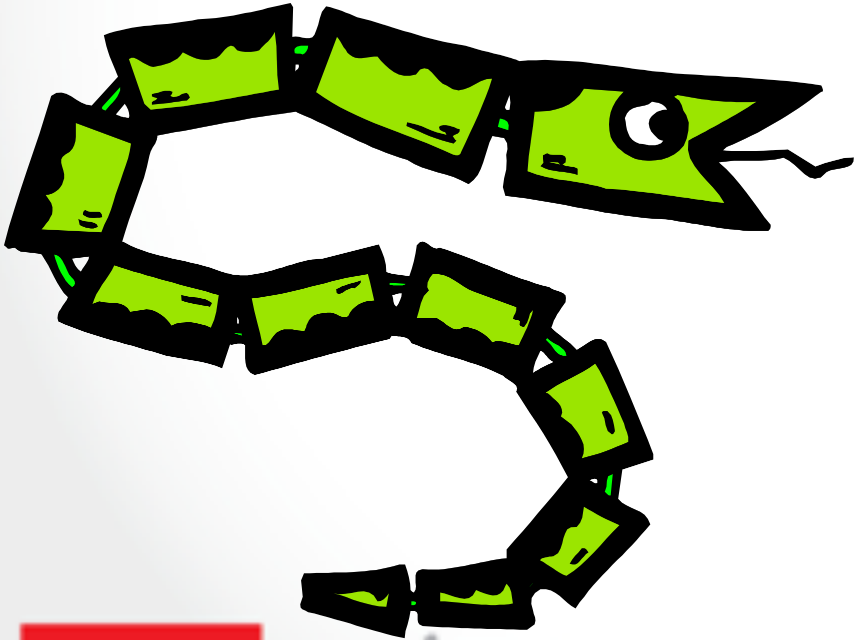
Simple functions: ex.py

```
"""factorial done recursively and iteratively"""

def fact1(n):
    ans = 1
    for i in range(2,n):
        ans = ans * n
    return ans

def fact2(n):
    if n < 1:
        return 1
    else:
        return n * fact2(n - 1)
```

Simple functions: ex.py

```
671> python
Python 2.5.2 …
>>> import ex
>>> ex.fact1(6)
1296
>>> ex.fact2(200)
78865786736479050355236321393218507…000000L
>>> ex.fact1
<function fact1 at 0x902470>
>>> fact1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fact1' is not defined
>>>
```

The Basics



A Code Sample (in IDLE)

```
x = 34 - 23              # A comment.
y = “Hello”              # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”   # String concat.
print x
print y
```

Enough to Understand the Code

- **Indentation matters to code meaning**
  - Block structure indicated by indentation
- **First assignment to a variable creates it**
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.
- **Assignment is = and comparison is ==**
- **For numbers + - * / % are as expected**
  - Special use of **+** for string concatenation and **%** for string formatting (as in C's printf)
- **Logical operators are words (`and, or, not`) *not* symbols**
- **The basic printing command is `print`**

Basic Datatypes

- **Integers (default for numbers)**

  z = 5 / 2 # Answer 2, integer division

- **Floats**

  x = 3.456

- **Strings**
  - Can use “” or ‘’ to specify with “abc” == ‘abc’
  - Unmatched can occur within the string: “matt’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them: “““a‘b“c”””

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines

- Use a newline to end a line of code

  Use \ when must go to next line prematurely

- No braces {} to mark blocks of code, use *consistent* indentation instead
  - First line with *less* indentation is outside of the block
  - First line with *more* indentation starts a nested block
- Colons start of a new block in many constructs, e.g. function definitions, then clauses

Comments

- Start comments with #, rest of line is ignored
- Can include a “documentation string” as the first line of a new function or class you define
- Development environments, debugger, and other tools use it: it’s good style to include one

```
def fact(n):
  “““fact(n) assumes n is a positive
  integer and returns facorial of n.”””
  assert(n>0)
   return 1 if n==1 else n*fact(n-1)
```

Assignment

- *Binding a variable* in Python means setting a *name* to hold a *reference* to some *object*
  - *Assignment creates references, not copies*
- Names in Python do not have an intrinsic type, objects have types
  - Python determines the type of the reference automatically based on what data is assigned to it
- You create a name the first time it appears on the left side of an assignment expression:

```
x = 3
```

- A reference is deleted via garbage collection after any names bound to it have passed out of scope
- Python uses *reference semantics* (more later)

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob   Bob   _bob   _2_bob_   bob_2   BoB
```

- There are some reserved words:

```
 and, assert, break, class, continue,
def, del, elif, else, except, exec,
finally, for, from, global, if,
import, in, is, lambda, not, or,
pass, print, raise, return, try,
while
```

Naming conventions

The Python community has these recommended naming conventions

- **joined_lower** for functions, methods and, attributes
- **joined_lower** or **ALL_CAPS** for constants
- **StudlyCaps** for classes
- **camelCase** only to conform to pre-existing conventions
- Attributes: interface, _internal, __private

Assignment

- You can assign to multiple names at the same time

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

This makes it easy to swap values

```
>>> x, y = y, x
```

- Assignments can be chained

```
>>> a = b = x = 2
```

Accessing Non-Existent Name

Accessing a name before it’s been properly created (by placing it on the left side of an assignment), raises an error

```
>>> y

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name ‘y' is not defined
>>> y = 3
>>> y
3
```

Sequence types: Tuples, Lists, and Strings

Sequence Types

1. **Tuple: (‘john’, 32, [CMSC])**
   - A simple *immutable* ordered sequence of items
   - Items can be of mixed types, including collection types
2. **Strings: “John Smith”**
   - *Immutable*
   - Conceptually very much like a tuple
3. **List: [1, 2, ‘john’, (‘up’, ‘down’)]**
   - *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
  - Tuples and strings are *immutable*
  - Lists are *mutable*
- The operations shown in this section can be applied to *all* sequence types
  - most examples will just show the operation performed on one

Sequence Types 1

- Define tuples using parentheses and commas

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Define lists are using square brackets and commas

```
>>> li = [“abc”, 34, 4.34, 23]
```

- Define strings using quotes (“, ‘, or “““).

```
>>> st = “Hello World”
>>> st = ‘Hello World’
>>> st = “““This is a multi-line
string that uses triple quotes.”””
```

Sequence Types 2

- Access individual members of a tuple, list, or string using square bracket “array” notation
- *Note that all are 0 based…*

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> tu[1]     # Second item in the tuple.
 ‘abc’

>>> li = [“abc”, 34, 4.34, 23]
>>> li[1]      # Second item in the list.
 34

>>> st = “Hello World”
>>> st[1]   # Second character in string.
 ‘e’
```

Positive and negative indices

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

Positive index: count from the left, starting with 0

```
>>> t[1]
‘abc’
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```

Slicing: return copy of a subset

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* second.

```
>>> t[1:4]
(‘abc’, 4.56, (2,3))
```

Negative indices count from end

```
>>> t[1:-1]
(‘abc’, 4.56, (2,3))
```

Slicing: return copy of a =subset

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

Omit first index to make copy starting from beginning of the container

```
    >>> t[:2]
    (23, ‘abc’)
```

Omit second index to make copy starting at first index and going to end

```
    >>> t[2:]
    (4.56, (2,3), ‘def’)
```

Copying the Whole Sequence

- [:] makes a *copy* of an entire sequence

```
>>> t[:]
(23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Note the difference between these two lines for mutable sequences

```
>>> l2 = l1 # Both refer to 1 ref,
            # changing one affects both
>>> l2 = l1[:] # Independent copies, two
 refs
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*

The + Operator

The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

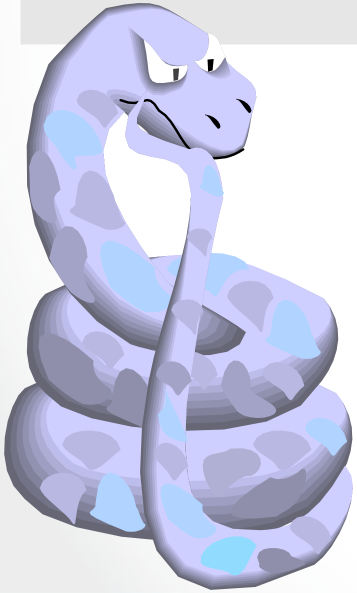
```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> “Hello” + “ ” + “World”
 ‘Hello World’
```

The * Operator

- The * operator produces a *new* tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

Mutability: Tuples vs. Lists



Lists are mutable

```
>>> li = [‘abc’, 23, 4.34, 23]
>>> li[1] = 45
>>> li
  [‘abc’, 45, 4.34, 23]
```

- We can change lists *in place.*
- Name *li* still points to the same memory reference when we’re done.

Tuples are immutable

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

- You can’t change a tuple.
- You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, ‘abc’, 3.14, (2,3), ‘def’)
```

- *The immutability of tuples means they’re faster than lists.*

Operations on Lists Only

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append(‘a’)  # Note the method
  syntax
>>> li
[1, 11, 3, 4, 5, ‘a’]

>>> li.insert(2, ‘i’)
>>>li
[1, 11, ‘i’, 3, 4, 5, ‘a’]
```

The *extend* method vs +

- + creates a fresh list with a new memory ref
- *extend* operates on list `li` in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, ‘i’, 3, 4, 5, ‘a’, 9, 8, 7]
```

- *Potentially confusing*:
  - *extend* takes a list as an argument.
  - *append* takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, ‘i’, 3, 4, 5, ‘a’, 9, 8, 7, [10,
   11, 12]]
```

Operations on Lists Only

Lists have many methods, including index, count, remove, reverse, sort

```
>>> li = [‘a’, ‘b’, ‘c’, ‘b’]
>>> li.index(‘b’)  # index of 1st occurrence
1
>>> li.count(‘b’)  # number of occurrences
2
>>> li.remove(‘b’) # remove 1st occurrence
>>> li
  [‘a’, ‘c’, ‘b’]
```

Operations on Lists Only

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()     # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

Tuple details

- The **comma** is the tuple creation operator, not parens

```
>>> 1,
(1,)
```

- Python shows parens for clarity (best practice)

```
>>> (1,)
(1,)
```

- Don't forget the comma!

```
>>> (1)
1
```

- Trailing comma only required for singletons others
- Empty tuples have a special syntactic form

```
>>> ()
()
>>> tuple()
()
```

Summary: Tuples vs. Lists

- Lists slower but more powerful than tuples
  - Lists can be modified, and they have lots of handy operations and mehtods
  - Tuples are immutable and have fewer features
- To convert between tuples and lists use the list() and tuple() functions:

```
li = list(tu)
tu = tuple(li)
```

Python features

Lutz, Programming Python

no compiling or linking	rapid development cycle
no type declarations	simpler, shorter, more flexible
automatic memory management	garbage collection
high-level data types and operations	fast development
object-oriented programming	code structuring and reuse, C++
embedding and extending in C	mixed language systems
classes, modules, exceptions	"programming-in-the-large" support
dynamic loading of C modules	simplified extensions, smaller binaries
dynamic reloading of C modules	programs can be modified without stopping

Python features

Lutz, Programming Python

universal "first-class" object model	fewer restrictions and rules
run-time program construction	handles unforeseen needs, end-user coding
interactive, dynamic nature	incremental development and testing
access to interpreter information	metaprogramming, introspective objects
wide portability	cross-platform programming without ports
compilation to portable byte-code	execution speed, protecting source code
built-in interfaces to external services	system tools, GUIs, persistence, databases, etc.

Uses of Python

- shell tools
  - system admin tools, command line programs
- extension-language work
- rapid prototyping and development
- language-based modules
  - instead of special-purpose parsers
- graphical user interfaces
- database access
- distributed programming
- Internet scripting

What not to use Python (and kin) for

- most scripting languages share these
- not as efficient as C
  - but sometimes better built-in algorithms (e.g., hashing and sorting)
- delayed error notification
- lack of profiling tools

Using python

- /usr/local/bin/python
  - #! /usr/bin/env python
- interactive use

```
Python 1.6 (#1, Sep 24 2000, 20:40:45)  [GCC 2.95.1 19990816 (release)] on sunos5
Copyright (c) 1995-2000 Corporation for National Research Initiatives.
All Rights Reserved.
Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>>
```

- python –c command [arg] ...
- python –i script
  - read script first, then interactive

Python structure

- modules: Python source files or C extensions
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
- objects
  - everything is an object
  - automatically reclaimed when no longer needed

Command line arguments

```
#!/usr/local/bin/python
# import systems module
import sys
marker = ':::::'
for name in sys.argv[1:]:
  input = open(name, 'r')
    print marker + name
  print input.read()
```

Basic operations

- Assignment:
  - size = 40
  - a = b = c = 3
- Numbers
  - integer, float
  - complex numbers: 1j+3, abs(z)
- Strings
  - 'hello world', 'it\'s hot'
  - "bye world"
  - continuation via \ or use """ long text """"

String operations

- concatenate with + or neighbors
  - word = 'Help' + x
  - word = 'Help' 'a'
- subscripting of strings
  - 'Hello'[2] → 'l'
  - slice: 'Hello'[1:2] → 'el'
  - word[-1] → last character
  - len(word) → 5
  - immutable: cannot assign to subscript

Lists

- lists can be heterogeneous
  - a = ['spam', 'eggs', 100, 1234, 2*2]
- Lists can be indexed and sliced:
  - a[0] → spam
  - a[:2] → ['spam', 'eggs']
- Lists can be manipulated
  - a[2] = a[2] + 23
  - a[0:2] = [1,12]
  - a[0:0] = []
  - len(a) → 5

Basic programming

```
a,b = 0, 1
# non-zero = true
while b < 10:
    # formatted output, without \n
  print b,
    # multiple assignment
  a,b = b, a+b
```

Control flow: if

```
x = int(raw_input("Please enter #:"))
if x < 0:
  x = 0
  print 'Negative changed to zero'
elif x == 0:
  print 'Zero'
elif x == 1:
  print 'Single'
else:
  print 'More'
```

- no case statement

Control flow: for

```
a = ['cat', 'window', 'defenestrate']
for x in a:
  print x, len(x)
```

- no arithmetic progression, but
  - range(10) → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

  -
    ```
    for i in range(len(a)):
                  print i, a[i]
    ```
- do not modify the sequence being iterated over

Loops: break, continue, else

- break and continue like C
- else after loop exhaustion

```
for n in range(2,10):
  for x in range(2,n):
    if n % x == 0:
      print n, 'equals', x, '*', n/x
      break
  else:
    # loop fell through without finding a factor
    print n, 'is prime'
```

Do nothing

- pass does nothing
- syntactic filler

```
while 1:
    pass
```

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

>>> fib(2000)
```

- First line is docstring
- first look for variables in local, then global
- need global to assign global variables

Functions: default argument values

```
def ask_ok(prompt, retries=4, complaint='Yes or no,
    please!'):
  while 1:
    ok = raw_input(prompt)
    if ok in ('y', 'ye', 'yes'): return 1
    if ok in ('n', 'no'): return 0
    retries = retries - 1
    if retries < 0: raise IOError, 'refusenik error'
    print complaint

>>> ask_ok('Really?')
```

Keyword arguments

- last arguments can be given as keywords

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian blue'):
  print "-- This parrot wouldn't", action,
  print "if you put", voltage, "Volts through it."
  print "Lovely plumage, the ", type
  print "-- It's", state, "!"

parrot(1000)
parrot(action='VOOOM', voltage=100000)
```

Lambda forms

- anonymous functions
- may not work in older versions

```
def make_incrementor(n):
  return lambda x: x + n

f = make_incrementor(42)
f(0)
f(1)
```

List methods

- append(x)
- extend(L)
  - append all items in list (like Tcl lappend)
- insert(i,x)
- remove(x)
- pop([i]), pop()
  - create stack (FIFO), or queue (LIFO) → pop(0)
- index(x)
  - return the index for value x

List methods

- count(x)
  - how many times x appears in list
- sort()
  - sort items in place
- reverse()
  - reverse list

Functional programming tools

- filter(function, sequence)

  ```
  def f(x): return x%2 != 0 and x%3 0
  filter(f, range(2,25))
  ```

- map(function, sequence)
  - call function for each item
  - return list of return values
- reduce(function, sequence)
  - return a single value
  - call binary function on the first two items
  - then on the result and next item
  - iterate

List comprehensions (2.0)

- Create lists without map(), filter(), lambda
- = expression followed by for clause + zero or more for or of clauses

```
>>> vec = [2,4,6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [{x: x**2} for x in vec}
[{2: 4}, {4: 16}, {6: 36}]
```

List comprehensions

- cross products:

```
>>> vec1 = [2,4,6]
>>> vec2 = [4,3,-9]
>>> [x*y for x in vec1 for y in vec2]
[8,6,-18, 16,12,-36, 24,18,-54]
>>> [x+y for x in vec1 and y in vec2]
[6,5,-7,8,7,-5,10,9,-3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8,12,-54]
```

List comprehensions

- can also use if:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

del - removing list items

- remove by index, not value
- remove slices from list (rather than by assigning an empty list)

```
>>> a = [-1,1,66.6,333,333,1234.5]
>>> del a[0]
>>> a
[1,66.6,333,333,1234.5]
>>> del a[2:4]
>>> a
[1,66.6,1234.5]
```

Tuples and sequences

- lists, strings, **tuples**: examples of sequence type
- tuple = values separated by commas

```
>>> t = 123, 543, 'bar'
>>> t[0]
123
>>> t
(123, 543, 'bar')
```

Tuples

- Tuples may be nested

```
>>> u = t, (1,2)
>>> u
((123, 542, 'bar'), (1,2))
```

- kind of like structs, but no element names:
  - (x,y) coordinates
  - database records
- like strings, immutable → can't assign to individual items

Tuples

- Empty tuples: ()

```
>>> empty = ()
>>> len(empty)
0
```

- one item → trailing comma

```
>>> singleton = 'foo',
```

Tuples

- sequence unpacking → distribute elements across variables

```
>>> t = 123, 543, 'bar'
>>> x, y, z = t
>>> x
123
```

- packing always creates tuple
- unpacking works for any sequence

Dictionaries

- like Tcl or awk associative arrays
- indexed by keys
- keys are any immutable type: e.g., tuples
- but not lists (mutable!)
- uses 'key: value' notation

```
>>> tel = {'hgs' : 7042, 'lennox': 7018}
>>> tel['cs'] = 7000
>>> tel
```

Dictionaries

- no particular order
- delete elements with del

```
>>> del tel['foo']
```

- keys() method → unsorted list of keys

```
>>> tel.keys()
['cs', 'lennox', 'hgs']
```

- use has_key() to check for existence

```
>>> tel.has_key('foo')
0
```

Conditions

- can check for sequence membership with is and is not:

```
>>> if (4 in vec):
...  print '4 is'
```

- chained comparisons: a less than b AND b equals c:

```
a < b == c
```

- and and or are short-circuit operators:
  - evaluated from left to right
  - stop evaluation as soon as outcome clear

Conditions

- Can assign comparison to variable:

```
>>> s1,s2,s3='', 'foo', 'bar'
>>> non_null = s1 or s2 or s3
>>> non_null
foo
```

- Unlike C, no assignment within expression

Comparing sequences

- unlike C, can compare sequences (lists, tuples, ...)
- lexicographical comparison:
  - compare first; if different → outcome
  - continue recursively
  - subsequences are smaller
  - strings use ASCII comparison
  - can compare objects of different type, but by type name (list < string < tuple)

Comparing sequences

```
(1,2,3) < (1,2,4)
[1,2,3] < [1,2,4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1,2,3) == (1.0,2.0,3.0)
(1,2) < (1,2,-1)
```

Modules

- collection of functions and variables, typically in scripts
- definitions can be imported
- file name is module name + .py
- e.g., create module fibo.py

```
def fib(n): # write Fib. series up to n
   ...
def fib2(n): # return Fib. series up to n
```

Modules

- import module:

```
import fibo
```

- Use modules via "name space":

```
>>> fibo.fib(1000)
>>> fibo.__name__
'fibo'
```

- can give it a local name:

```
>>> fib = fibo.fib
>>> fib(500)
```

Modules

- function definition + executable statements
- executed only when module is imported
- modules have private symbol tables
- avoids name clash for global variables
- accessible as module.globalname
- can import into name space:

```
>>> from fibo import fib, fib2
>>> fib(500)
```

- can import all names defined by module:

```
>>> from fibo import *
```

Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable
- uses installation-default if not defined, e.g., .:/usr/local/lib/python
- uses sys.path

```
>>> import sys
>>> sys.path
['', 'C:\\PROGRA~1\\Python2.2', 'C:\\Program Files\\Python2.2\\DLLs', 'C:\\Program
    Files\\Python2.2\\lib', 'C:\\Program Files\\Python2.2\\lib\\lib-tk', 'C:\\Program Files\
    \Python2.2', 'C:\\Program Files\\Python2.2\\lib\\site-packages']
```

Compiled Python files

- include byte-compiled version of module if there exists fibo.pyc in same directory as fibo.py
- only if creation time of fibo.pyc matches fibo.py
- automatically write compiled file, if possible
- platform independent
- doesn't run any faster, but loads faster
- can have only .pyc file → hide source

Standard modules

- system-dependent list
- always sys module

```
>>> import sys
>>> sys.p1
'>>> '
>>> sys.p2
'... '
>>> sys.path.append('/some/directory')
```

Module listing

- use dir() for each module

```
>>> dir(fibo)
['___name___', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__', '__st
din__', '__stdout__', '_getframe', 'argv', 'builtin_module_names', 'byteorder',
'copyright', 'displayhook', 'dllhandle', 'exc_info', 'exc_type', 'excepthook', '
exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getrecursionlimit', '
getrefcount', 'hexversion', 'last_type', 'last_value', 'maxint', 'maxunicode', '
modules', 'path', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setpr
ofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
 'version_info', 'warnoptions', 'winver']
```

Classes

- mixture of C++ and Modula-3
- multiple base classes
- derived class can override any methods of its base class(es)
- method can call the method of a base class with the same name
- objects have private data
- C++ terms:
  - all class members are public
  - all member functions are virtual
  - no constructors or destructors (not needed)

Classes

- classes (and data types) are objects
- built-in types cannot be used as base classes by user
- arithmetic operators, subscripting can be redefined for class instances (like C++, unlike Java)

Class definitions

```
Class ClassName:
  <statement-1>
  ...
  <statement-N>
```

- must be executed
- can be executed conditionally (see Tcl)
- creates new namespace

Namespaces

- mapping from name to object:
  - built-in names (abs())
  - global names in module
  - local names in function invocation
- attributes = any following a dot
  - z.real, z.imag
- attributes read-only or writable
  - module attributes are writeable

Namespaces

- scope = textual region of Python program where a namespace is directly accessible (without dot)
  - innermost scope (first) = local names
  - middle scope = current module's global names
  - outermost scope (last) = built-in names
- assignments always affect innermost scope
  - don't copy, just create name bindings to objects
- global indicates name is in global scope

Class objects

- obj.name references (plus module!):

```
class MyClass:
  "A simple example class"
  i = 123
  def f(self):
    return 'hello world'
>>> MyClass.i
123
```

- MyClass.f is method object

Class objects

- class instantiation:

```
>>> x = MyClass()
>>> x.f()
'hello world'
```

- creates new instance of class
  - note x = MyClass vs. x = MyClass()
- ____init___() special method for initialization of object

```
def __init__(self,realpart,imagpart):
        self.r = realpart
        self.i = imagpart
```

Instance objects

- attribute references
- data attributes (C++/Java data members)
  - created dynamically

```
x.counter = 1
while x.counter < 10:
  x.counter = x.counter * 2
print x.counter
del x.counter
```

Method objects

- Called immediately:

```
x.f()
```

- can be referenced:

```
xf = x.f
while 1:
  print xf()
```

- object is passed as first argument of function → 'self'
  - x.f() is equivalent to MyClass.f(x)

Notes on classes

- Data attributes override method attributes with the same name
- no real hiding → not usable to implement pure abstract data types
- clients (users) of an object can add data attributes
- first argument of method usually called self
    - 'self' has **no** special meaning (cf. Java)

Another example

- bag.py

```
class Bag:
  def __init__(self):
    self.data = []
  def add(self, x):
    self.data.append(x)
  def addtwice(self,x):
    self.add(x)
    self.add(x)
```

Another example, cont'd.

- invoke:

```
>>> from bag import *
>>> l = Bag()
>>> l.add('first')
>>> l.add('second')
>>> l.data
['first', 'second']
```

Inheritance

```
class DerivedClassName(BaseClassName)
  <statement-1>
  ...
  <statement-N>
```

- search class attribute, descending chain of base classes
- may override methods in the base class
- call directly via BaseClassName.method

Multiple inheritance

```
class DerivedClass(Base1,Base2,Base3):
    <statement>
```

- depth-first, left-to-right
- problem: class derived from two classes with a common base class

Private variables

- No real support, but textual replacement (name mangling)
- __var is replaced by _classname_var
- prevents only accidental modification, not true protection

~ C structs

- Empty class definition:

```
class Employee:
    pass

john = Employee()
john.name = 'John Doe'
john.dept = 'CS'
john.salary = 1000
```

Exceptions

- syntax (parsing) errors

```
while 1 print 'Hello World'
File "<stdin>", line 1
while 1 print 'Hello World'
              ^
SyntaxError: invalid syntax
```

- exceptions
    - run-time errors
    - e.g., ZeroDivisionError, NameError, TypeError

Handling exceptions

```
while 1:
  try:
    x = int(raw_input("Please enter a number: "))
    break
  except ValueError:
    print "Not a valid number"
```

- First, execute try clause
- if no exception, skip except clause
- if exception, skip rest of try clause and use except clause
- if no matching exception, attempt outer try statement

Handling exceptions

- try.py

```
import sys
for arg in sys.argv[1:]:
  try:
  f = open(arg, 'r')
  except IOError:
  print 'cannot open', arg
  else:
  print arg, 'lines:', len(f.readlines())
  f.close
```

- e.g., as python try.py *.py

Language comparison

		Tcl	Perl	Python	JavaScript	Visual Basic
Speed	development	✓	✓	✓	✓	✓
	regexp	✓	✓	✓		
breadth	extensible	✓		✓		✓
	embeddable	✓		✓		
	easy GUI	✓		✓ (Tk)		✓
	net/web	✓	✓	✓	✓	✓
enterprise	cross-platform	✓	✓	✓	✓	
	I18N	✓		✓	✓	✓
	thread-safe	✓		✓		✓
	database access	✓	✓	✓	✓	✓

Text and File Processing

Strings

- **string**: A sequence of text characters in a program.
  - Strings start and end with quotation mark `"` or apostrophe `'` characters.
  - Examples:

```
"hello"
"This is a string"
"This, too, is a string.   It can be very long!"
```

- A string may not span across multiple lines or contain a " character.

```
"This is not
a legal String."
 "This is not a "legal" String either."
```

- A string can represent characters by preceding them with a backslash.
  - `\t` tab character
  - `\n` new line character
  - `\"` quotation mark character
  - `\\` backslash character
  - Example: `"Hello\tthere\nHow are you?"`

Indexes

- Characters in a string are numbered with *indexes* starting at 0:
  - Example:

```
name = "P. Diddy"
```

index	0	1	2	3	4	5	6	7
character	P	.		D	i	d	d	y

- Accessing an individual character of a string:

  variableName [***index***]

  - Example:

```
print name, "starts with", name[0]
```

Output:

```
P. Diddy starts with P
```

String properties

- `len(`***string***`)` - number of characters in a string (including spaces)
- `str.lower(`***string***`)` - lowercase version of a string
- `str.upper(`***string***`)` - uppercase version of a string

- Example:

```
name = "Martin Douglas Stepp"
length = len(name)
big_name = str.upper(name)
print big_name, "has", length, "characters"
```

Output:

```
MARTIN DOUGLAS STEPP has 20 characters
```

raw_input

- `raw_input` : Reads a string of text from user input.
  - Example:

```
name = raw_input("Howdy, pardner. What's yer name? ")
print name, "... what a silly name!"
```

Output:

```
Howdy, pardner. What's yer name? Paris Hilton
Paris Hilton ... what a silly name!
```

Text processing

- **text processing**: Examining, editing, formatting text.
  - often uses loops that examine the characters of a string one by one

- A `for` loop can examine each character in a string in sequence.
  - Example:

```
for c in "booyah":
    print c
```

Output:

```
b
o
o
y
a
h
```

Strings and numbers

- `ord(`***text***`)` - converts a string into a number.
  - Example: `ord("a")` is `97`, `ord("b")` is `98`, ...
  - Characters map to numbers using standardized mappings such as *ASCII* and *Unicode*.
- `chr(`***number***`)` - converts a number into a string.
  - Example: `chr(99)` is `"c"`
- **Exercise:** Write a program that performs a rotation cypher.
  - e.g. `"Attack"` when rotated by 1 becomes `"buubdl"`

File processing

- Many programs handle data, which often comes from files.

- Reading the entire contents of a file:

  variableName = open("***filename***").read()

  Example:

  ```
  file_text = open("bankaccount.txt").read()
  ```

Line-by-line processing

- Reading a file line-by-line:

```
for line in open("filename").readlines():
    statements
```

Example:

```
count = 0
for line in open("bankaccount.txt").readlines():
    count = count + 1
print "The file contains", count, "lines."
```

- **Exercise:** Write a program to process a file of DNA text, such as:

```
ATGCAATTGCTCGATTAG
```

  - Count the percent of C+G present in the DNA.

Graphics

DrawingPanel

- To create a window, create a `drawingpanel` and its graphical pen, which we'll call `g` :

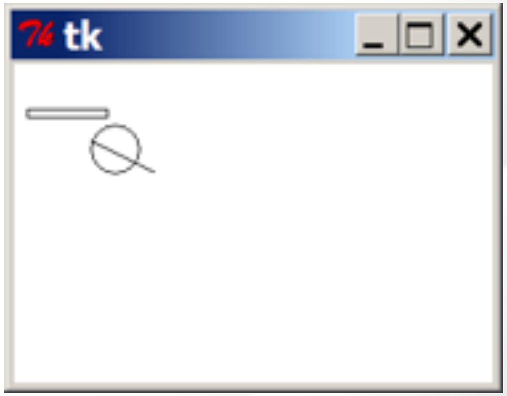
```
from drawingpanel import *
panel = drawingpanel(width, height)
g = panel.get_graphics()
```

... (draw shapes here) ...

```
panel.mainloop()
```

- The window has nothing on it, but we can draw shapes and lines on it by sending commands to `g` .
  - Example:

```
g.create_rectangle(10, 30, 60, 35)
g.create_oval(80, 40, 50, 70)
g.create_line(50, 50, 90, 70)
```

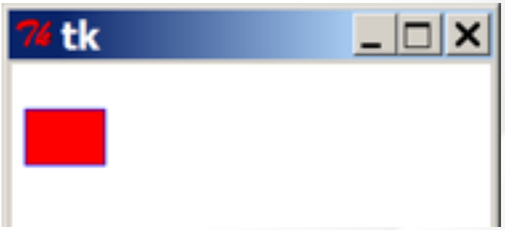


Graphical commands

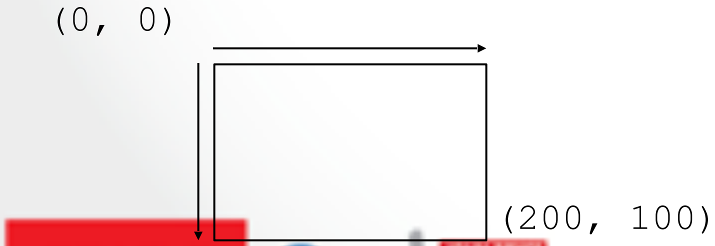
Command	Description
g.create_line(***x1***, ***y1***, ***x2***, ***y2***)	a line between (***x1***, ***y1***), (***x2***, ***y2***)
g.create_oval(***x1***, ***y1***, ***x2***, ***y2***)	the largest oval that fits in a box with top-left corner at (***x1***, ***y1***) and bottom-left corner at (***x2***, ***y2***)
g.create_rectangle(***x1***, ***y1***, ***x2***, ***y2***)	the rectangle with top-left corner at (***x1***, ***y1***), bottom-left at (***x2***, ***y2***)
g.create_text(***x***, ***y***, text="***text***")	the given ***text*** at (***x***, ***y***)

- The above commands can accept optional outline and fill colors.

```
g.create_rectangle(10, 40, 22, 65, fill="red", outline="blue")
```



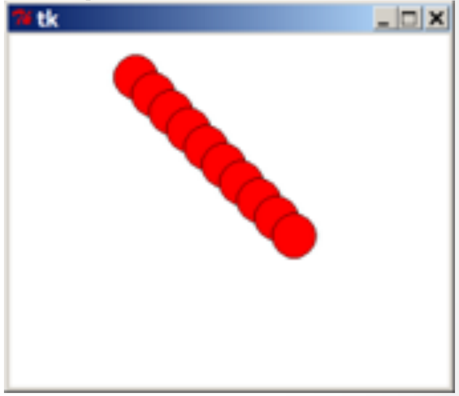
- The coordinate system is y-inverted:



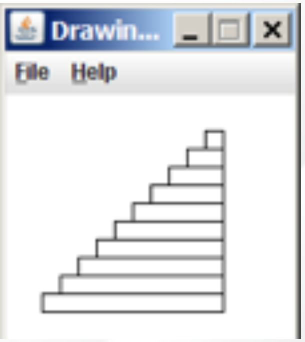
Drawing with loops

- We can draw many repetitions of the same item at different x/y positions with `for` loops.
  - The x or y assignment expression contains the loop counter, `i`, so that in each pass of the loop, when `i` changes, so does x or y.

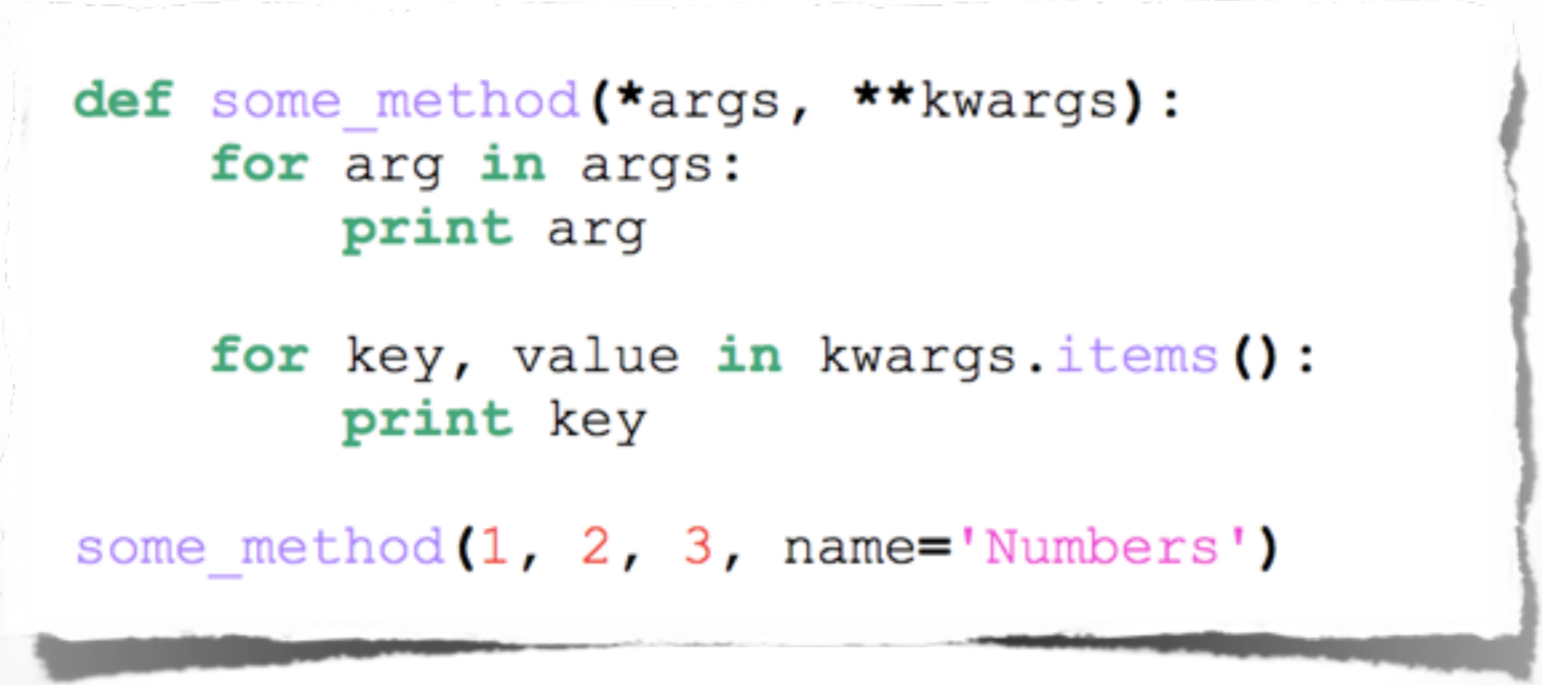
```
from drawingpanel import *

window = drawingpanel(500, 400)
g = window.get_graphics()

for i in range(1, 11):
    x = 100 + 20 * i
    y = 5 + 20 * i
    g.create_oval(x, y, x + 50, y + 50, fill="red")

window.mainloop()
```



- **Exercise:** Draw the figure at right.

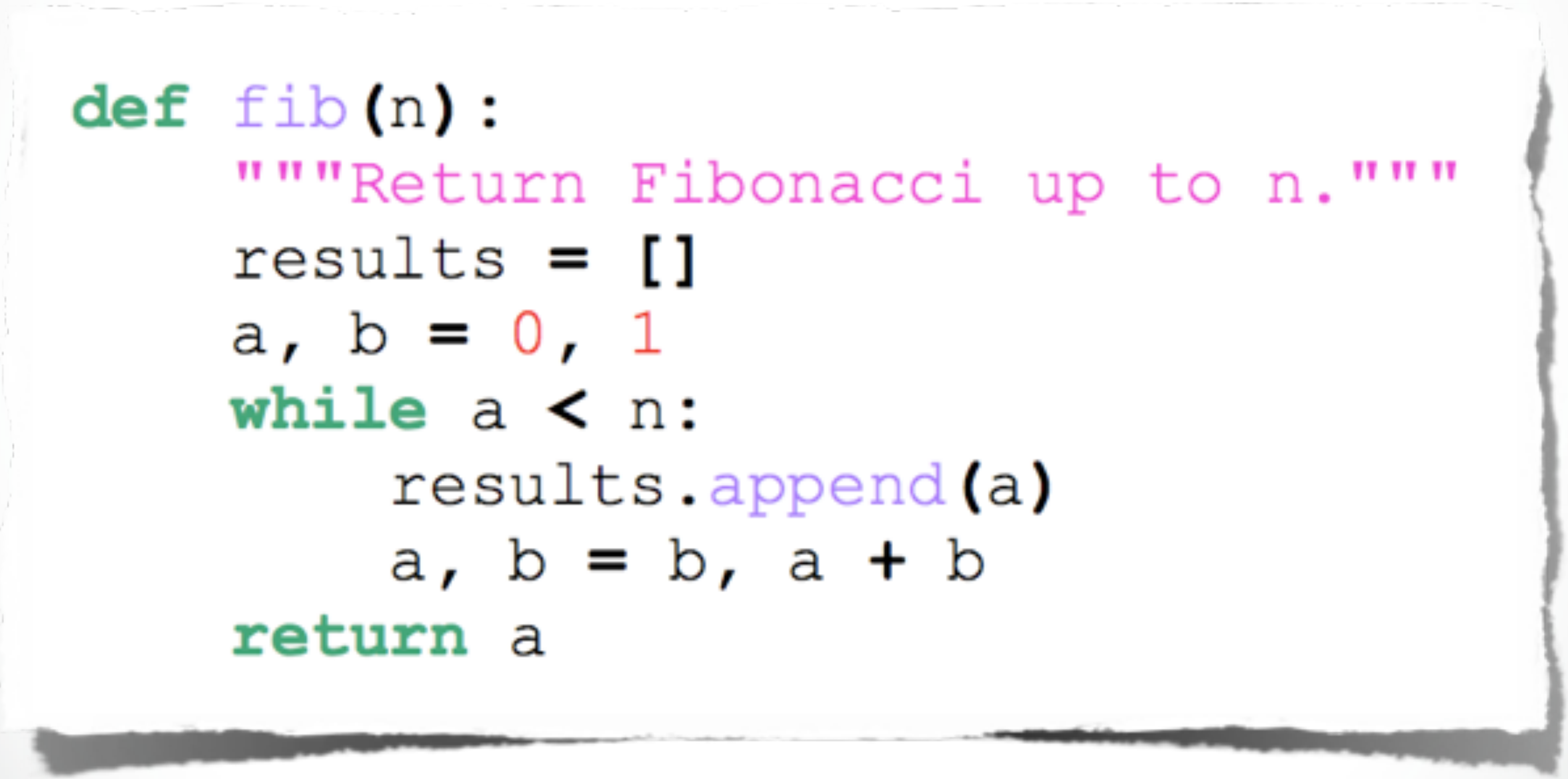


Arbitrary Arguments



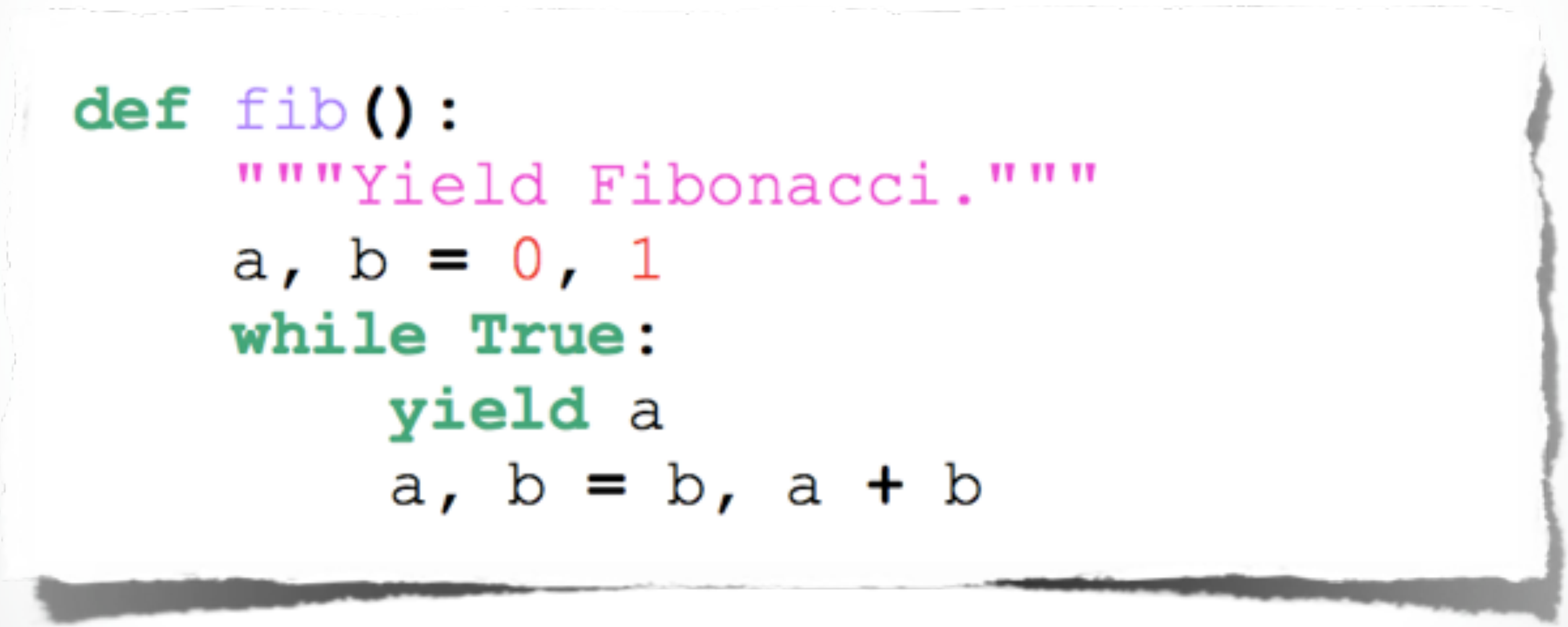
```
def some_method(*args, **kwargs):
    for arg in args:
        print arg

    for key, value in kwargs.items():
        print key

some_method(1, 2, 3, name='Numbers')
```

Fibonacci



```
def fib(n):
    """Return Fibonacci up to n."""
    results = []
    a, b = 0, 1
    while a < n:
        results.append(a)
        a, b = b, a + b
    return a
```

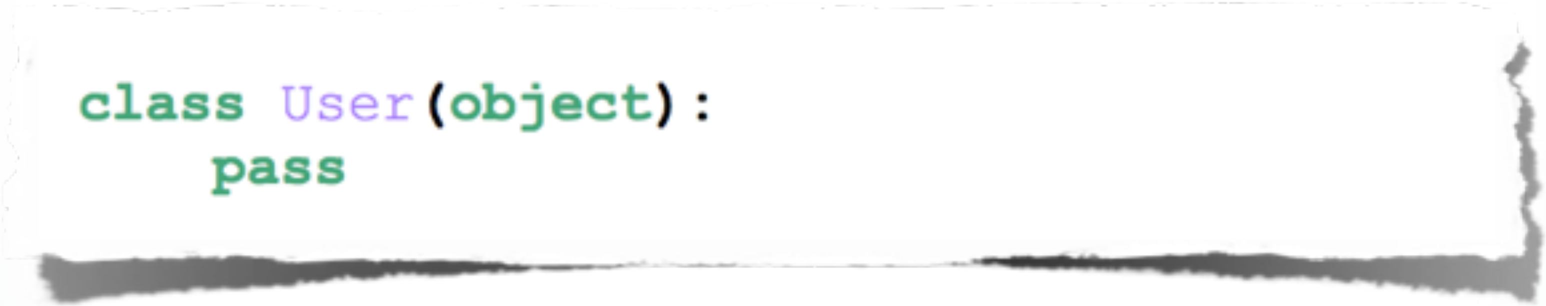
Fibonacci Generator



```python
def fib():
    """Yield Fibonacci."""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

Classes

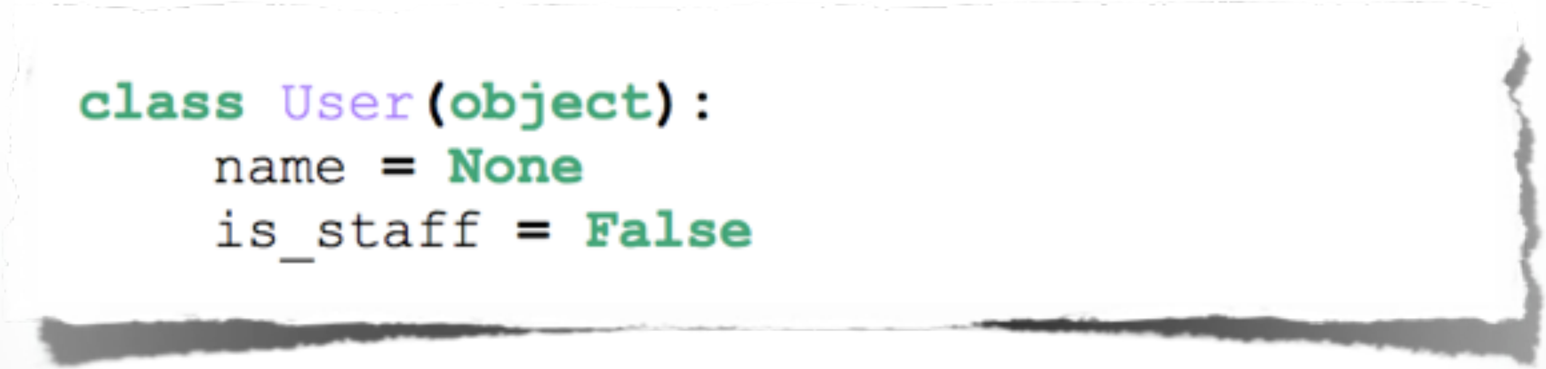
Class Declaration



```
class User(object):
    pass
```

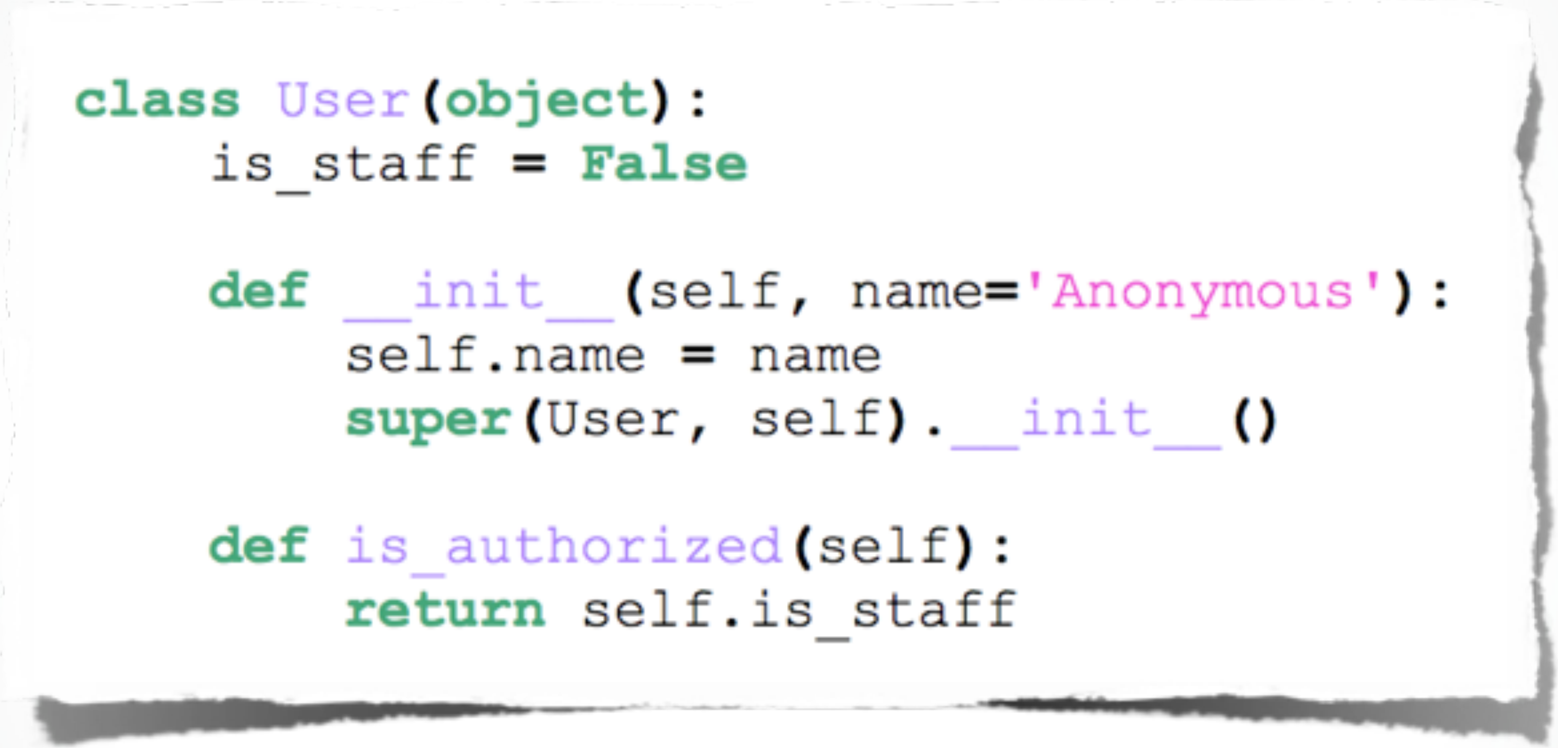
Class Attributes

- Attributes assigned at class declaration should always be immutable



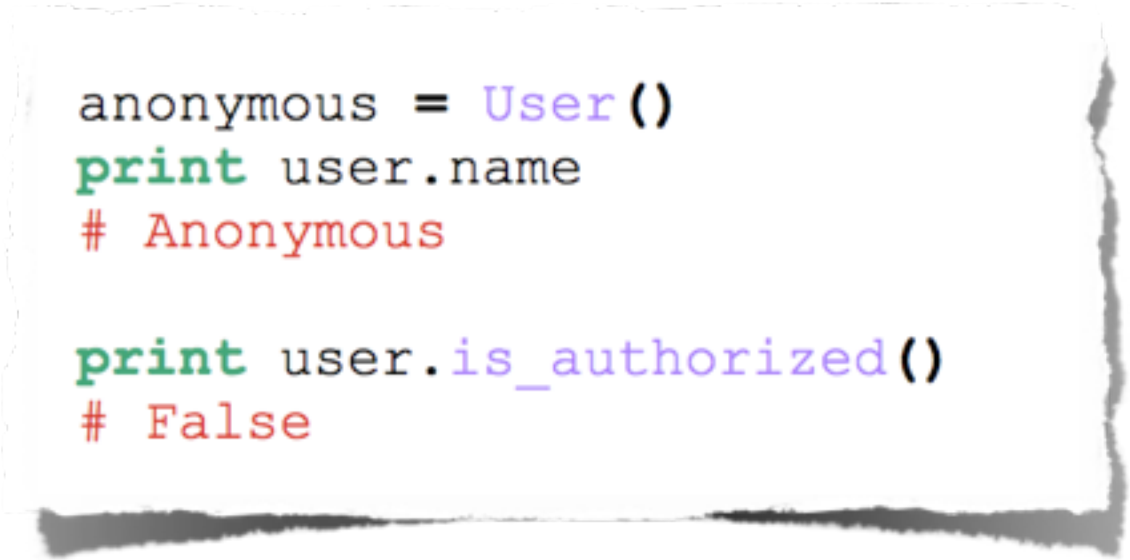
```
class User(object):
    name = None
    is_staff = False
```

Class Methods



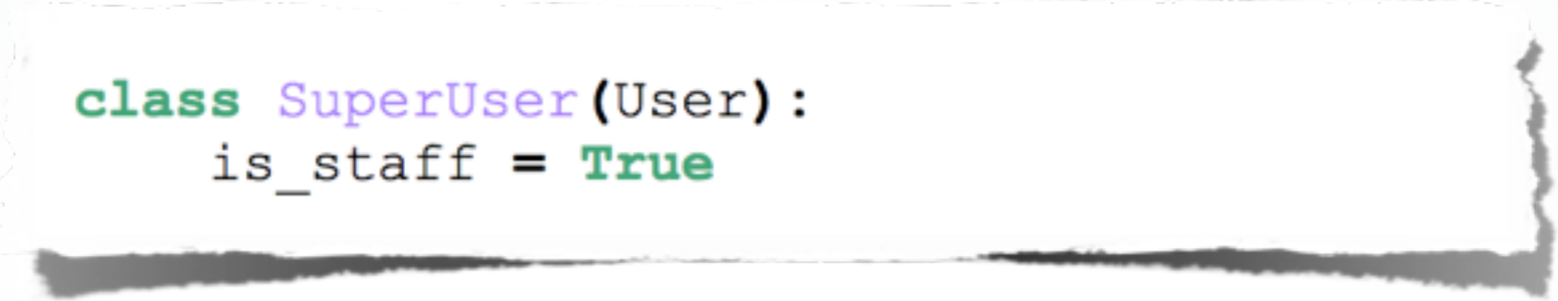
```
class User(object):
    is_staff = False

    def __init__(self, name='Anonymous'):
        self.name = name
        super(User, self).__init__()

    def is_authorized(self):
        return self.is_staff
```

Class Instantiation & Attribute Access

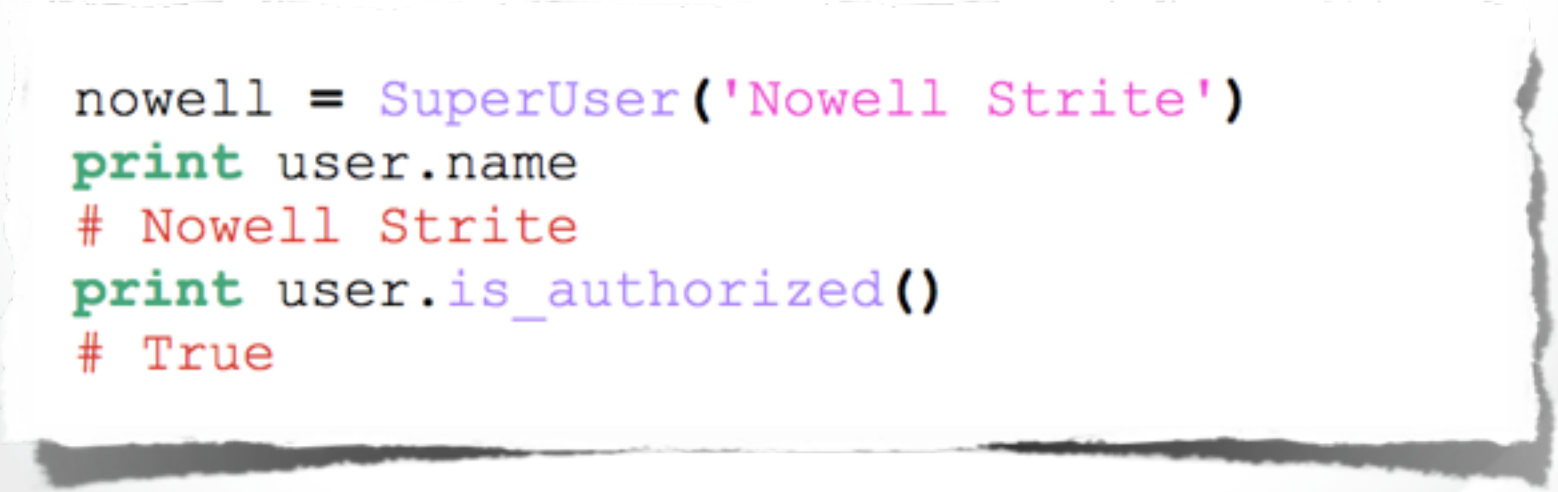


```
anonymous = User()
print user.name
# Anonymous

print user.is_authorized()
# False
```

Class Inheritance



```
class SuperUser(User):
    is_staff = True
```



```
nowell = SuperUser('Nowell Strite')
print user.name
# Nowell Strite
print user.is_authorized()
# True
```

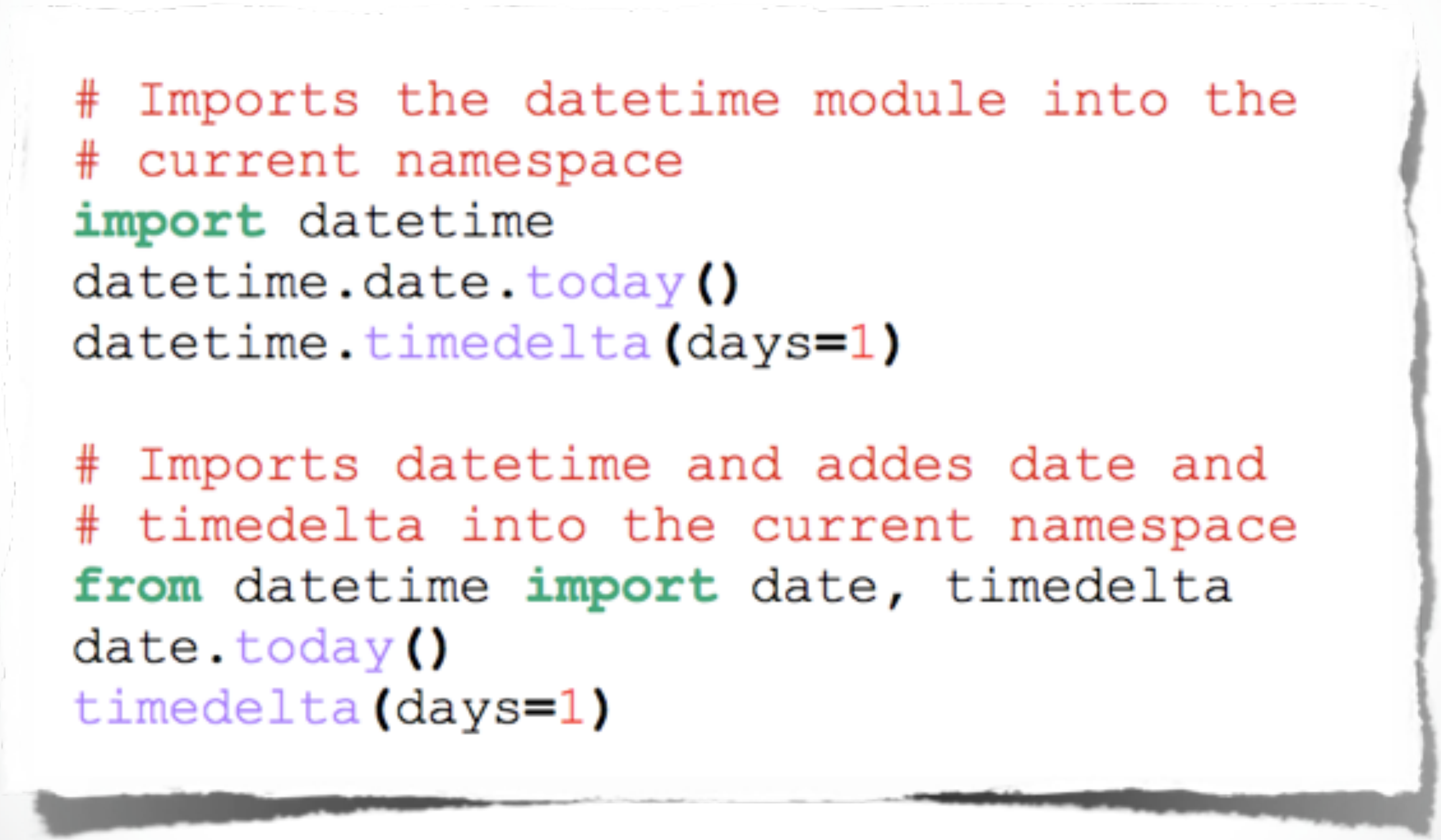
Python’s Way

- No interfaces
- No real private attributes/functions
- Private attributes start (but do not end) with double underscores.
- Special class methods start and end with double underscores.
    - __init__, __doc__, __cmp__, __str__

Imports

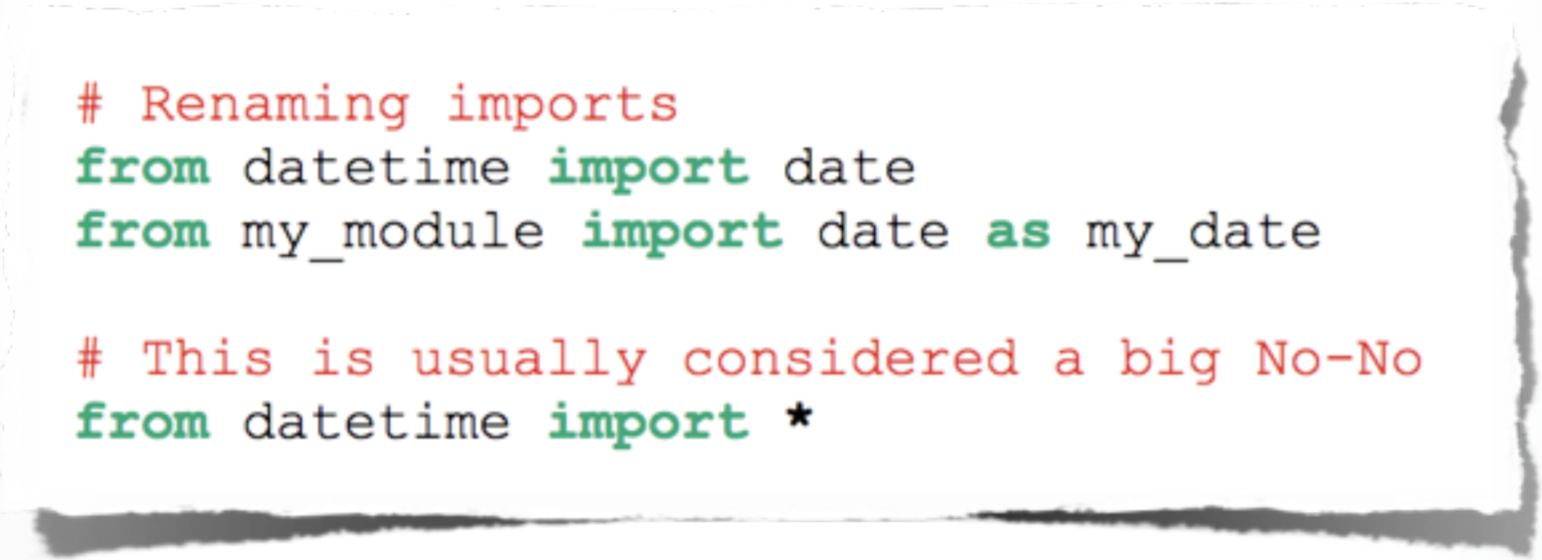
- Allows code isolation and re-use
- Adds references to variables/classes/functions/etc. into current namespace

Imports



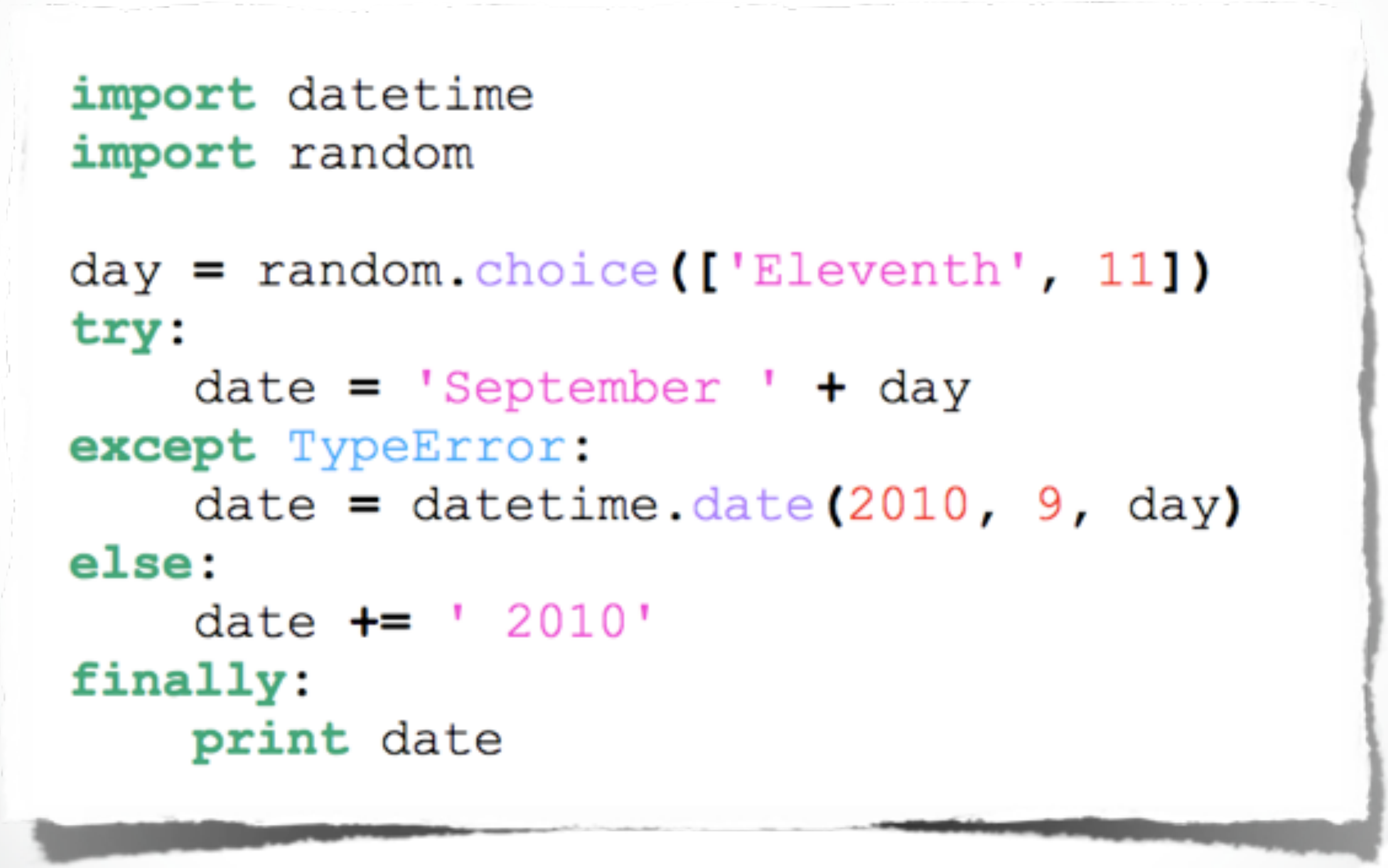
```
# Imports the datetime module into the
# current namespace
import datetime
datetime.date.today()
datetime.timedelta(days=1)

# Imports datetime and addes date and
# timedelta into the current namespace
from datetime import date, timedelta
date.today()
timedelta(days=1)
```

More Imports



```
# Renaming imports
from datetime import date
from my_module import date as my_date

# This is usually considered a big No-No
from datetime import *
```

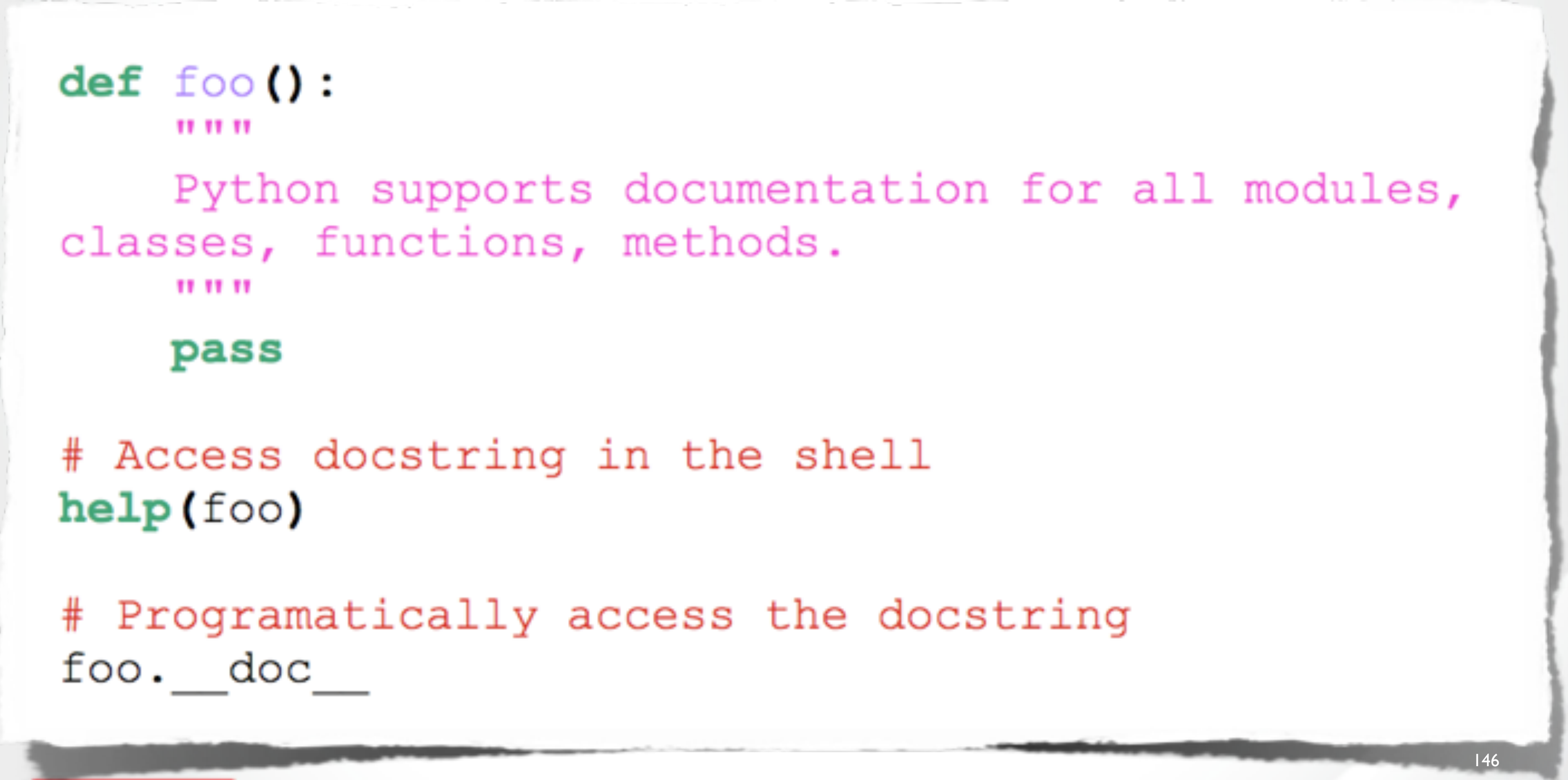
Error Handling



```
import datetime
import random

day = random.choice(['Eleventh', 11])
try:
    date = 'September ' + day
except TypeError:
    date = datetime.date(2010, 9, day)
else:
    date += ' 2010'
finally:
    print date
```

Documentation

Docstrings



```
def foo():
    """
    Python supports documentation for all modules,
classes, functions, methods.
    """
    pass

# Access docstring in the shell
help(foo)

# Programatically access the docstring
foo.__doc__
```

Tools

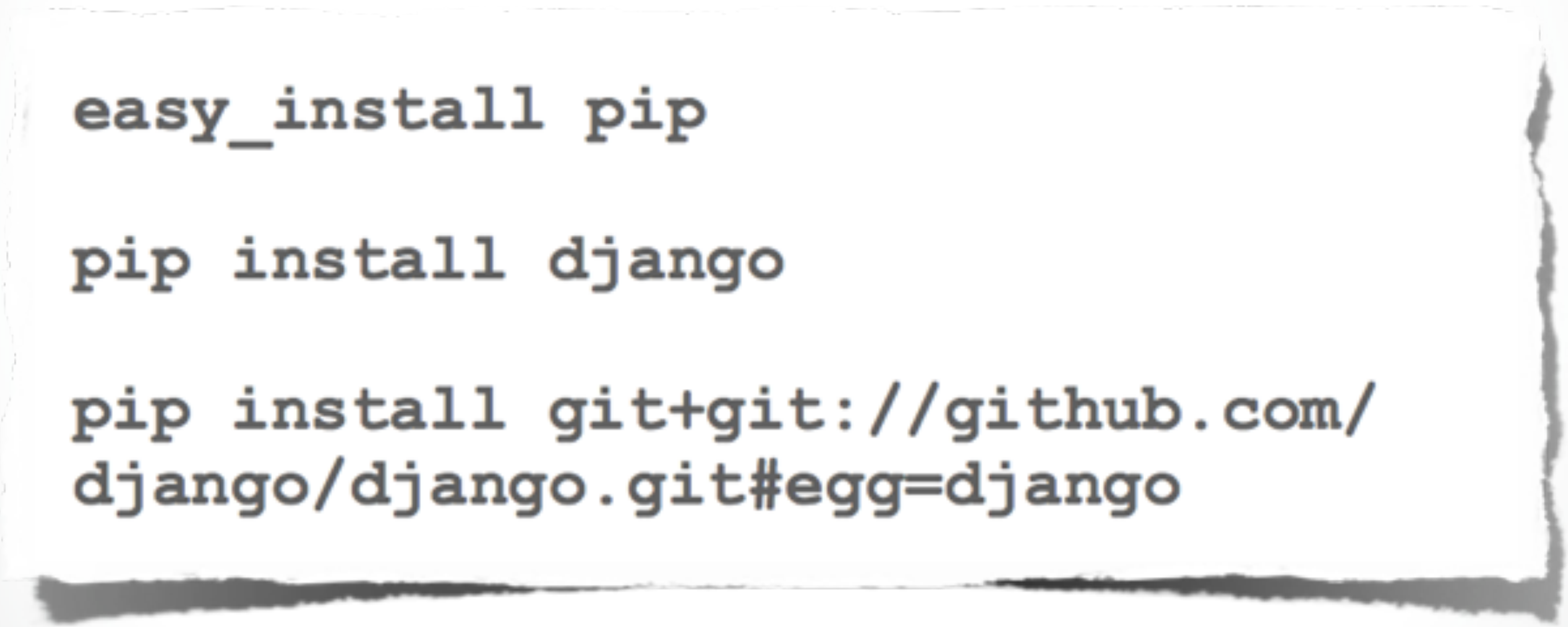
Web Frameworks

- Django
- Flask
- Pylons
- TurboGears
- Zope
- Grok

IDEs

- Emacs
- Vim
- Komodo
- PyCharm
- Eclipse (PyDev)

Package Management

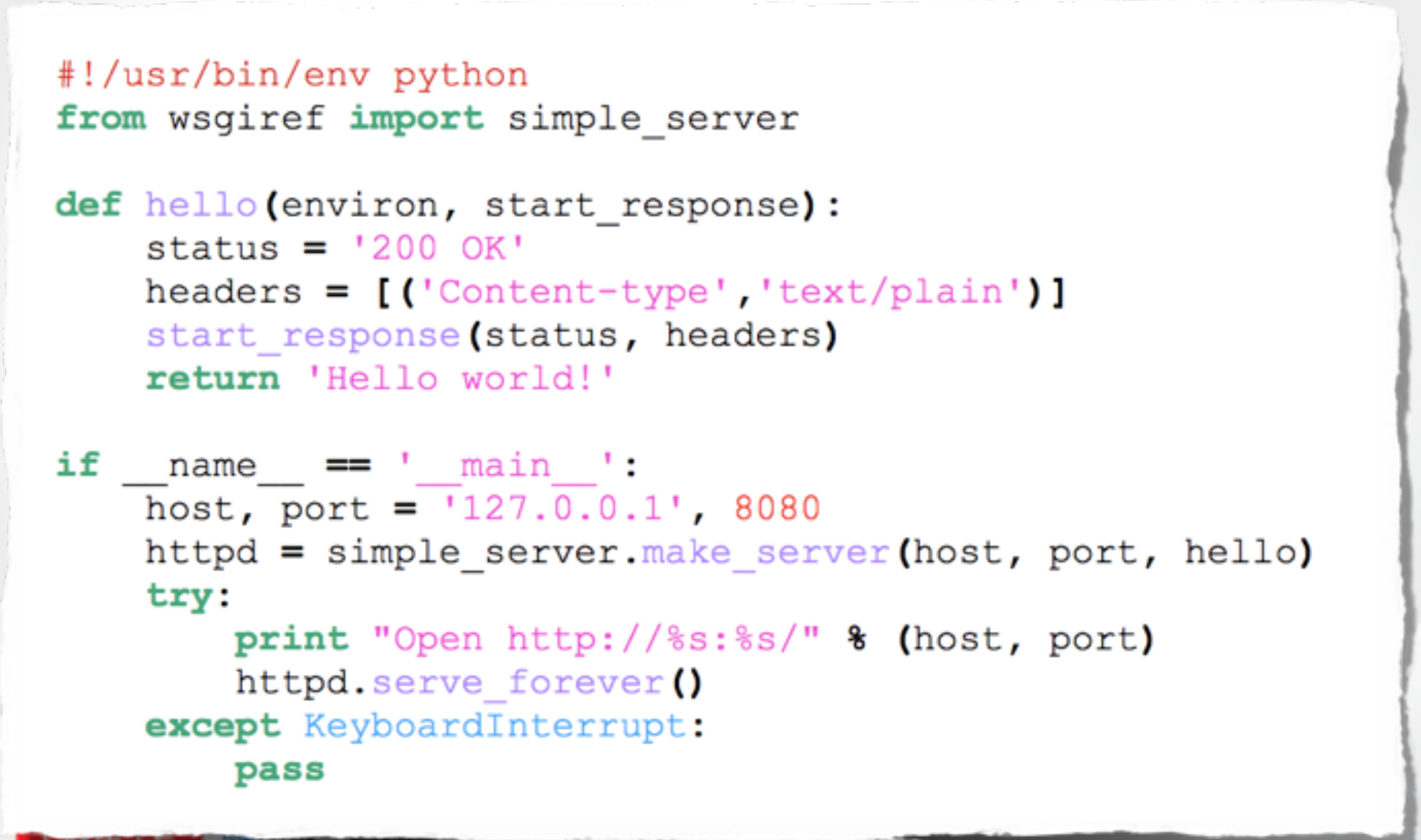


```
easy_install pip

pip install django

pip install git+git://github.com/
django/django.git#egg=django
```

Resources

- http://python.org/
- http://diveintopython.org/
- http://djangoproject.com/

Example



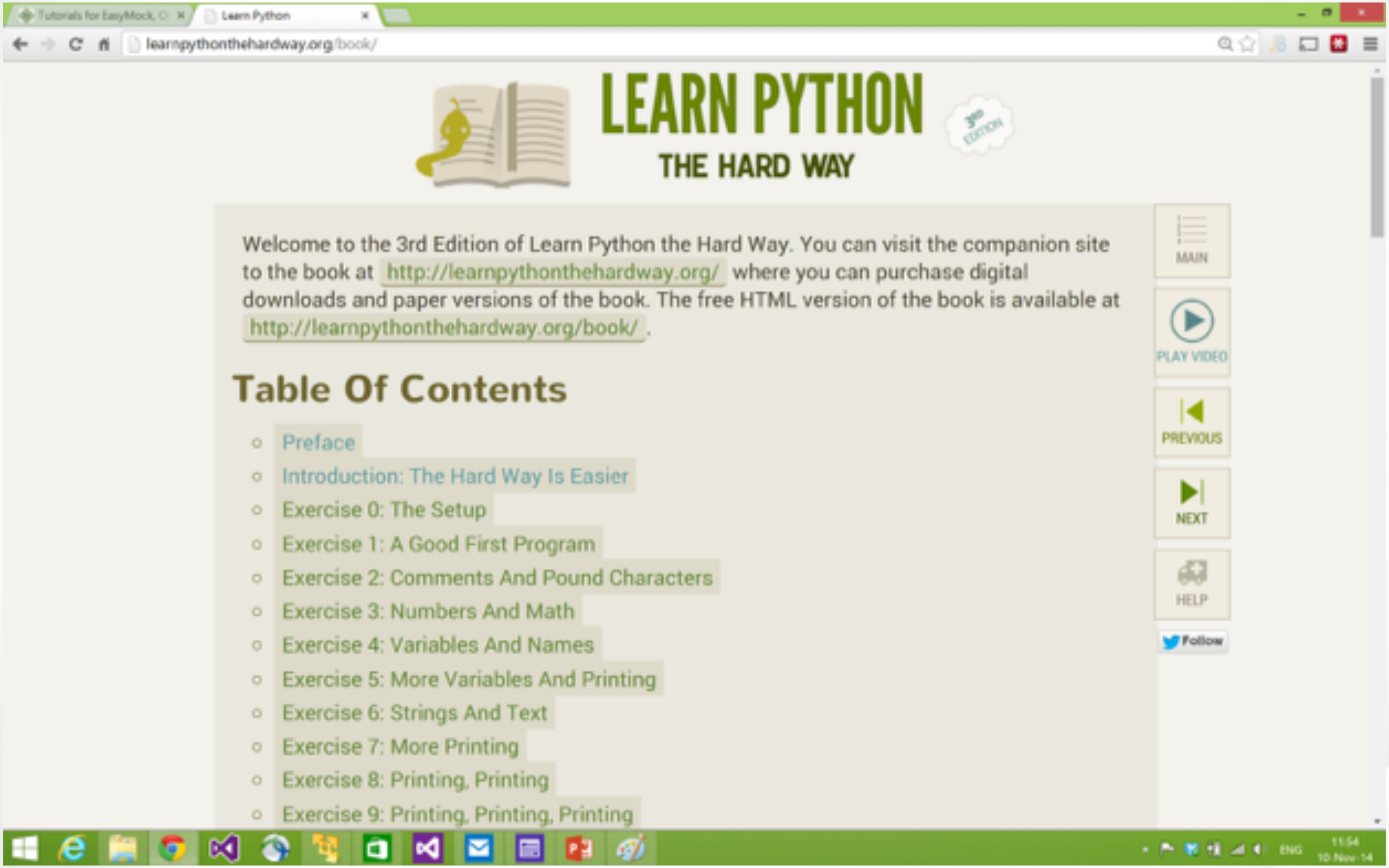
```
#!/usr/bin/env python
from wsgiref import simple_server

def hello(environ, start_response):
    status = '200 OK'
    headers = [('Content-type','text/plain')]
    start_response(status, headers)
    return 'Hello world!'

if __name__ == '__main__':
    host, port = '127.0.0.1', 8080
    httpd = simple_server.make_server(host, port, hello)
    try:
        print "Open http://%s:%s/" % (host, port)
        httpd.serve_forever()
    except KeyboardInterrupt:
        pass
```

Going Further

- Decorators
- Context Managers
- Lambda functions
- Generators
- ...

Questions?

Learn Python The Hard Way



Welcome to the 3rd Edition of Learn Python the Hard Way. You can visit the companion site to the book at http://learnpythonthehardway.org/ where you can purchase digital downloads and paper versions of the book. The free HTML version of the book is available at http://learnpythonthehardway.org/book/ .

Table Of Contents

שבוע אורקל 2014

Thank You!