

Practical Python Programming

David M. Beazley
<http://www.dabeaz.com>

Edition: Fri May 8 11:41:56 2015

Copyright (C) 2010-2015
David M Beazley
All Rights Reserved

Practical Python Programming : Table of Contents

0. Course Setup	1
1. Introduction	6
2. Working with Data	51
3. Program Organization and Functions	85
4. Modules and Libraries	108
5. Classes	173
6. Inside the Python Object Model	192
7. Documentation, Testing, and Debugging	215
8. Iterators and Generators	234
9. Working with Text	249
10. Some Advanced Topics	266

Edition: Fri May 8 11:41:56 2015

Course Summary

This course is an introduction to Python primarily designed for programmers, scientists, or engineers who already know another programming language, but are new to Python. Various features of Python are introduced with a focus on using Python for various tasks involving scripting, data analysis, and file processing. By the end of this course, you will have an overview of most of Python's core features and know how to write short programs.

System Requirements

The course assumes the use of Python 2.7 on any operating system platform. An optional part of section 4 (modules) involves using numpy (<http://numpy.scipy.org>), Pandas (<http://pandas.pydata.org/>) and matplotlib (<http://matplotlib.sourceforge.net>).

Support Files and Exercises

Support files and exercises must be downloaded at the following URL:

<http://www.dabeaz.com/python/practicalpython.zip>

This zip file needs to be extracted on your machine. You will find course exercises in the practical-python/Exercises folder.

0. Course Setup

Course Setup	0-1
Required Files	0-2
Setting up Your Environment	0-3
Class Exercises	0-4
Solution Code	0-5
General Tips	0-6
Text Editors	0-7
Running IDLE (Windows)	0-8
Running IDLE (Mac/Unix)	0-9
Alternate Tools	0-10

1. Introduction to Python

Introduction to Python	1-1
What is Python?	1-2
Where to Get Python?	1-3
Python Versions	1-4
Why was Python Created?	1-5
Some Uses of Python	1-6
Python Non-Uses	1-7
Running Python	1-8
IDLE	1-9
IDLE on Windows	1-10
IDLE on other Systems	1-11
The Python Interpreter	1-12
Interactive Mode	1-13
Getting Help	1-16
Exercise 1.1	1-17
Creating Programs	1-18
Running Programs (IDLE)	1-22
Running Programs	1-23
A Sample Program	1-24
Exercise 1.2	1-27
Python 101 : Statements	1-28
Python 101 : Comments	1-29
Python 101: Variables	1-30
Python 101: Case Sensitivity	1-31
Python 101: Cleaning up	1-32
Python 101: Looping	1-33
Python 101 : Indentation	1-34
Python 101 : Conditionals	1-37
Python 101 : Relations	1-38
Python 101 : Truth Values	1-39
Python 101 : Printing	1-40
Python 101 : User Input	1-41
Python 101 : pass statement	1-42
Python 101 : Long Lines	1-43
Exercise 1.3	1-44
Basic Datatypes	1-45
Numbers	1-46
Booleans	1-47
Integers	1-48
Integer Operations	1-49
Integer Division	1-50

Floating point (float)	1-51
Floating point	1-52
Floating Point Operators	1-53
Converting Numbers	1-54
Strings	1-55
String Escape Codes	1-56
String Representation	1-57
More String Operations	1-59
String Methods	1-60
More String Methods	1-61
String Mutability	1-62
String Conversions	1-63
Special Strings	1-64
Exercise 1.4	1-65
String Splitting	1-66
Lists	1-67
Lists (cont)	1-68
More List Operations	1-69
List Searching	1-70
List Removal	1-71
List Iteration	1-72
List Sorting	1-73
Lists and Math	1-74
Exercise 1.5	1-75
File Input and Output	1-76
Reading File Data	1-77
Writing To a File	1-79
File Management	1-80
Exercise 1.6	1-81
Type Conversion	1-82
Simple Functions	1-83
Library Functions	1-84
Exception Handling	1-85
Exceptions	1-86
Summary	1-88
Exercise 1.7	1-89

2. Working with Data

Working with Data	2-1
Overview	2-2
Primitive Datatypes	2-3
None type	2-4
Data Structures	2-5
Tuples	2-6
Tuple Use	2-7
Tuples (cont)	2-8
Tuple Packing	2-9
Tuple Unpacking	2-10
Tuple Commentary	2-11
Dictionaries	2-12
Exercise 2.1	2-15
Containers	2-16
Lists as a Container	2-17
List Construction	2-18
Dicts as a Container	2-19
Dict Construction	2-20

Dictionary Lookups	2-21
Sets	2-22
Set Example	2-23
Exercise 2.2	2-24
Formatted Output	2-25
String Formatting	2-26
Format Codes	2-27
format() Function	2-28
format() method	2-29
Exercise 2.3	2-30
Working with Sequences	2-31
Sequence Slicing	2-33
Extended Slices	2-34
Sequence Reductions	2-35
Iterating over a Sequence	2-36
Iteration Variables	2-37
break statement	2-38
continue statement	2-39
Looping over integers	2-40
Caution with range()	2-41
enumerate() Function	2-42
for and tuples	2-44
zip() Function	2-45
Exercise 2.4	2-46
List Comprehensions	2-47
List Comp: Examples	2-50
Historical Digression	2-51
List Comp. and Awk	2-52
Exercise 2.5	2-53
More details on objects	2-54
The Issue with Assignment	2-55
Assignment Example	2-56
Reassigning Values	2-58
Some Dangers	2-59
Identity and References	2-60
Shallow Copies	2-61
Deep Copying	2-62
Names, Values, Types	2-63
Type Checking	2-64
Everything is an object	2-65
First Class Objects	2-66
Summary	2-67
Exercise 2.6	2-68

3. Program Organization and Function

Program Organization and Function	3-1
Overview	3-2
Observation	3-3
What is a "Script?"	3-4
Problem	3-5
Defining Things	3-6
Defining Functions	3-7
What is a function?	3-8
Function Definitions	3-9
Bottom-up Style	3-10

Function Arguments	3-11
Function Design	3-12
Exercise 3.1	3-13
Default Arguments	3-14
Calling a Function	3-15
Keyword Arguments	3-16
Design Tip	3-17
Return Values	3-18
Multiple Return Values	3-19
Understanding Variables	3-20
Local Variables	3-21
Global Variables	3-22
Modifying Globals	3-23
Argument Passing	3-25
Understanding Assignment	3-26
Exercise 3.2	3-27
Error Checking	3-28
Exceptions	3-30
Builtin-Exceptions	3-34
Exception Values	3-35
Catching Multiple Errors	3-36
Catching All Errors	3-37
Exploding Heads	3-38
A Better Approach	3-39
Reraising an Exception	3-40
Exception Advice	3-41
finally statement	3-42
with statement	3-43
Program Exit	3-44
Exercise 3.3	3-45

4. Modules and Libraries

Modules and Libraries	4-1
Overview	4-2
Modules	4-3
Namespaces	4-4
Global Definitions	4-5
Modules as Environments	4-6
Module Execution	4-7
import as statement	4-8
from module import	4-9
from module import *	4-10
Be Explicit	4-12
Commentary	4-13
Main Functions	4-14
Main Module	4-15
__main__ check	4-16
Module Loading	4-18
Locating Modules	4-19
Module Search Path	4-20
Exercise 4.1	4-21
Standard Library	4-22
sys module	4-23
sys: Standard I/O	4-24
sys: Command Line Opts	4-26
Advanced Arguments	4-27

6. The Inner Workings of Python Objects

The Inner Workings of Python Obj	6-1
Overview	6-2
Dictionaries Revisited	6-3
Dicts and Modules	6-4
Dicts and Objects	6-5
Dicts and Instances	6-6
Dicts and Classes	6-8
Instances and Classes	6-9
Attribute Access	6-11
Modifying Instances	6-12
Reading Attributes	6-14
Exercise 6.1	6-16
How Inheritance Works	6-17
Reading Attributes	6-18
Single Inheritance	6-19
The MRO	6-20
Multiple Inheritance	6-21
Why super()?	6-26
Some Cautions	6-27
Classes and Encapsulation	6-28
A Problem	6-29
Python Encapsulation	6-30
Private Attributes	6-31
Problem: Simple Attributes	6-34
Managed Attributes	6-35
Properties	6-36
Uniform Access	6-41
Decorator Syntax	6-42
Properties and Accessors	6-43
__slots__ Attribute	6-44
Commentary	6-45
Exercise 6.2	6-46

7. Testing and Debugging

Testing and Debugging	7-1
Overview	7-2
Testing Rocks, Debugging Sucks	7-3
Testing Modules	7-4
Testing: doctest module	7-5
Using doctest	7-6
Doctest Caution	7-8
unittest Module	7-9
Using unittest	7-10
Running unittests	7-13
unittest comments	7-14
Third Party Test Tools	7-15
Exercise 7.1	7-16
logging Module	7-17
Exceptions Revisited	7-18

Using Logging	7-20
Logging Basics	7-21
Logging Configuration	7-22
Big Picture	7-23
Exercise 7.2	7-24
Assertions	7-25
Contract Programming	7-26
Optimized mode	7-27
__debug__ variable	7-28
Error Handling	7-29
The Python Debugger	7-30
Python Debugger	7-32
Remote Debugging	7-34
Profiling	7-35
Profile Sample Output	7-36
Summary	7-37
Exercise 7.3	7-38

8. Generators

Generators	8-1
Iteration	8-2
Iteration Everywhere	8-3
Iteration: Protocol	8-4
Supporting Iteration	8-6
The itertools Module	8-7
Exercise 8.1	8-8
Customizing Iteration	8-9
Generators	8-10
Generator Functions	8-11
Exercise 8.2	8-14
Producers & Consumers	8-15
Generator Pipelines	8-16
Exercise 8.3	8-21
Generator Expressions	8-22
Exercise 8.4	8-25
Why Use Generators?	8-26
The itertools Module	8-29
More Information	8-30

9. Working with Text

Text I/O Handling	9-1
Overview	9-2
Generating Text	9-3
String Concatenation	9-4
String Joining	9-5
String Joining Example	9-6
String Interpolation	9-7
Built-in Formatting	9-8
Template Strings	9-9
Exercise 9.1	9-10
Text Input/Output	9-11
Line Handling	9-12
Universal Newline	9-14
Text Encoding	9-16

International Characters	9-17	Advanced Topics	10-49
Unicode	9-18	Shameless Plug	10-50
Unicode Characters	9-19		
Unicode Charts	9-20		
Using Unicode Charts	9-21		
Unicode Representation	9-23		
Unicode I/O	9-24		
Unicode File I/O	9-25		
Unicode Encoding	9-26		
Encoding Errors	9-27		
Finding the Encoding	9-30		
Unicode Everywhere	9-31		
A Caution	9-32		
Exercise 9.2	9-33		

10. A Few Advanced Topics

A Few Advanced Topics	10-1
Overview	10-2
Variable Arguments	10-3
Passing Tuples and Dicts	10-6
Exercise 10.1	10-7
List Sorting Revisited	10-8
List Sorting	10-9
Callback Functions	10-11
Anonymous Functions	10-12
Using lambda	10-13
lambda and map()	10-14
Advice on Lambda	10-15
Exercise 10.2	10-16
Returning Functions	10-17
Local Variables	10-18
Closures	10-19
Using Closures	10-21
Delayed Evaluation	10-22
Exercise 10.3	10-24
Function Decorators	10-25
An Example	10-26
Observation	10-27
An Example	10-28
Decorators	10-30
Using Decorators	10-31
Commentary	10-32
Exercise 10.4	10-33
Decorated Methods	10-34
Static Methods	10-35
Using Static Methods	10-36
Class Methods	10-37
Using Class Methods	10-38
Exercise 10.5	10-40
Packages	10-41
Creating a Package	10-42
Using a Package	10-44
__init__.py files	10-45
Package Issues	10-46
Exercise 10.6	10-47
That's All Folks!	10-48

Section 0

Course Setup

Required Files

- Where to get Python (if not installed)

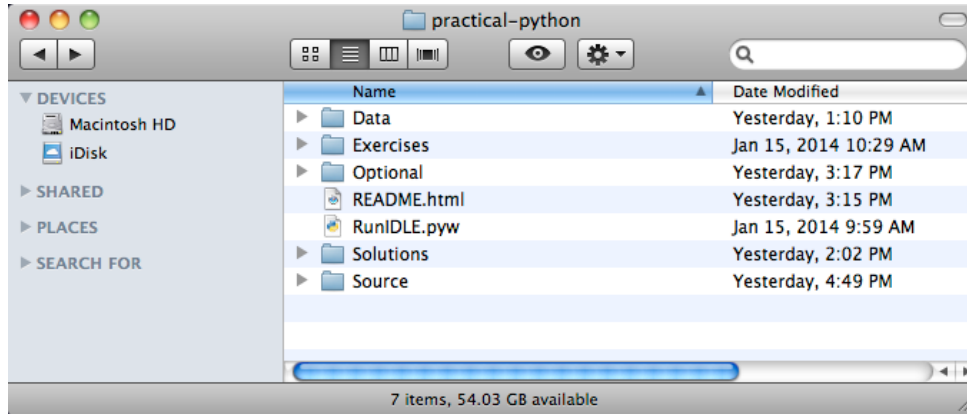
<http://www.python.org>

- Exercises for this class

<http://www.dabeaz.com/python/practicalpython.zip>

Setting up Your Environment

- Extract practical-python on your machine



- This folder is where you will do your work

Class Exercises

- Exercise descriptions are found in [practical-python/Exercises/index.html](http://practical-python.com/exercises/index.html)

- All exercises have solution code

One nice thing about the `csv` module is that it deals with a variety of output, you'll notice that it has stripped the double-quotes away from the output. Modify your `pcost.py` program so that it uses the `csv` module for parsing the output.



Look for the link at the bottom!

Solution Code

- Fully working solution code is found in [practical-python/Solutions/](#)
- It's okay to look at it, copy it, etc.
- However, I encourage you to try and come up with your own solution first.

General Tips

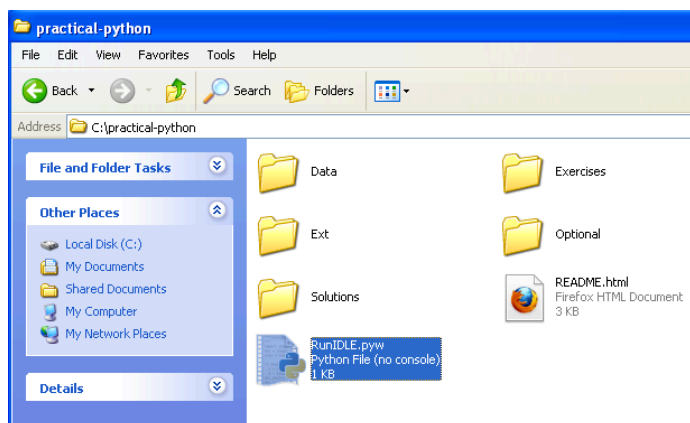
- We will be writing a lot of programs that access data files in `practical-python/Data`
- Save your programs in the `"practical-python/"` directory so that the names of these files are easy to access.
- Some exercises are more difficult than others. Please copy code from the solution and study it if necessary.

Text Editors

- You may use any text editor that you are familiar with for this course (e.g., vim, emacs, etc.)
- Python also comes with a simple development environment called IDLE.
- It's not the most advanced tool, but it works
- Follow the instructions on the next two slides to start it in the correct environment

Running IDLE (Windows)

- Find RunIDLE in the practical-python/ folder
- Double-click to start the IDLE environment



Running IDLE (Mac/Unix)

- Go into the practical-python/ directory
- Type the following command in a command shell

```
% python -m idlelib.idle
```
- Note: Typing 'idle' at the shell might also work.

Note: IDLE has known issues on the Mac. You may need to install a new version of Python to use it effectively.

Alternate Tools

- Some free alternatives to IDLE
 - PyCharm
 - PyScripter
 - Komodo Edit
 - Eclipse with Pydev
 - Wing IDE
- Search on Google for more details

Section I

Introduction to Python

What is Python?

- An interpreted high-level programming language.
- Similar to Perl, Ruby, Tcl, and other so-called "scripting languages."
- Created by Guido van Rossum around 1990.
- Named in honor of Monty Python

Where to Get Python?

<http://www.python.org>

- Downloads
- Documentation and tutorial
- Community Links
- Third party packages
- News and more

Python Versions

- Most users use "CPython"
 - Version 2.X (most common)
 - Version 3.X (bleeding edge, the future)
- Alternative implementations
 - Jython
 - IronPython
 - PyPy

Why was Python Created?

"My original motivation for creating Python was the perceived [need for a higher level language](#) in the Amoeba [Operating Systems] project. I realized that the [development of system administration utilities](#) in C was taking too long. Moreover, doing these things in the Bourne shell wouldn't work for a variety of reasons. ... So, there was a need for [a language that would bridge the gap between C and the shell.](#)"

- Guido van Rossum

Some Uses of Python

- Text processing/data processing
- Application scripting
- Systems administration/programming
- Internet programming
- Graphical user interfaces
- Testing
- Writing quick "throw-away" code

Python Non-Uses

- Device drivers and low-level systems
- Computer graphics, visualization, and games
- Numerical algorithms

Comment : Python is still used in these application domains, but only as a high-level control language. Important computations are actually carried out in C, C++, Fortran, etc. For example, you would not implement matrix-multiplication in Python.

Running Python

- Python programs run inside an interpreter
- The interpreter is a simple "console-based" application that normally starts from a command shell (e.g., the Unix shell)

```
bash % python
Python 2.7.3 (r271:86832, Feb 27 2011, 11:47:28)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license"
>>>
```

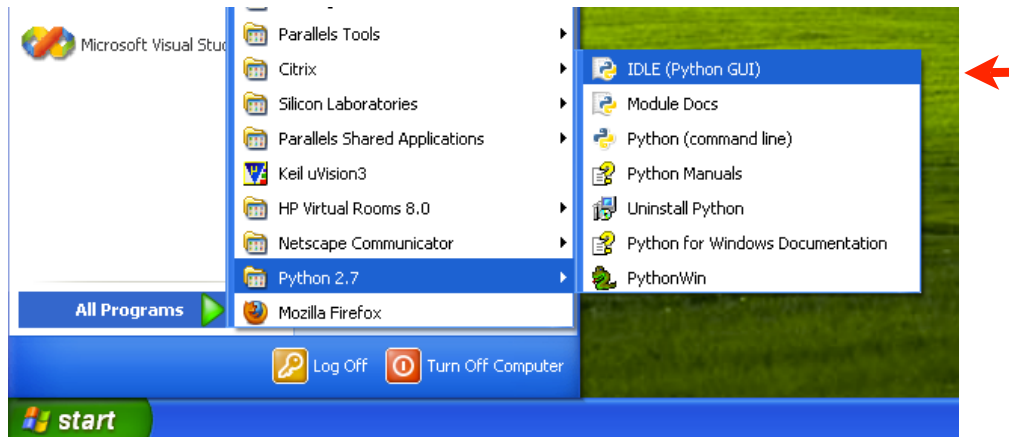
- Expert programmers usually have no problem using the interpreter in this way, but it's not so user-friendly for beginners

IDLE

- Python includes a simple integrated development called IDLE (which is another Monty Python reference)
- It's not the most sophisticated environment but it's already installed and it works
- Most working Python programmers tend to use something else, but it is fine for this class.

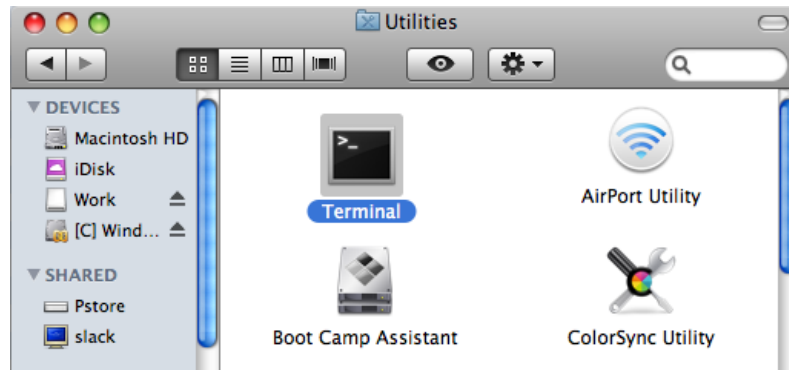
IDLE on Windows

- Look for it in the "Start" menu



IDLE on other Systems

- Launch a terminal or command shell



- Type the following command to launch IDLE

```
bash % python -m idlelib.idle
```

The Python Interpreter

- When you start Python, you get an "interactive" mode where you can experiment
- If you start typing statements, they will run immediately
- No edit/compile/run/debug cycle

Interactive Mode

- The interpreter runs a "read-eval" loop

```
>>> print "hello world"
hello world
>>> 37*42
1554
>>> for i in range(5):
...     print i
...
0
1
2
3
4
>>>
```

- Executes simple statements typed in directly
- Very useful for debugging, exploration

Interactive Mode

- Some notes on using the interactive shell

>>> is the interpreter prompt for starting a new statement

... is the interpreter prompt for continuing a statement (it may be blank in some tools)

Enter a blank line to finish typing and to run

```
>>> print "hello world"
hello world
>>> 37*42
1554
>>> for i in range(5):
...     print i
...
0
1
2
3
4
>>>
```

Interactive Mode

- Use underscore (`_`) for the last result

```
>>> 37*42
1554
>>> _ * 2
3108
>>> _ + 50
3158
>>>
```

- Note: This only works in interactive mode (you never use `_` in a program)

Getting Help

- `help(name)` command

```
>>> help(range)
Help on built-in function range in module __builtin__:

range(...)
    range([start,] stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!)
    ...
>>>
```

- Type `help()` with no name for interactive help
- Documentation at <http://docs.python.org>

Exercise 1.1

Time: 10 minutes

Creating Programs

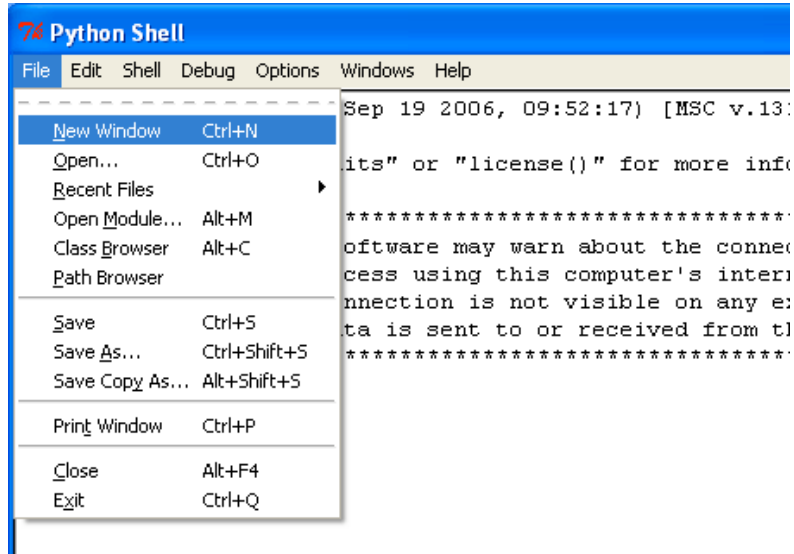
- Programs are put in .py files

```
# helloworld.py  
print "hello world"
```

- Source files are simple text files
- Create with your favorite editor (e.g., emacs)
- Can also edit programs with IDLE or other Python IDE (too many to list)

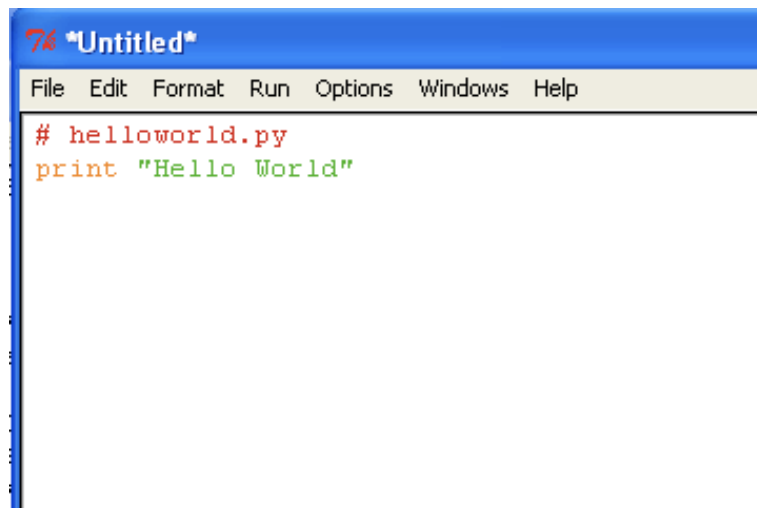
Creating Programs

- Creating a new program in IDLE



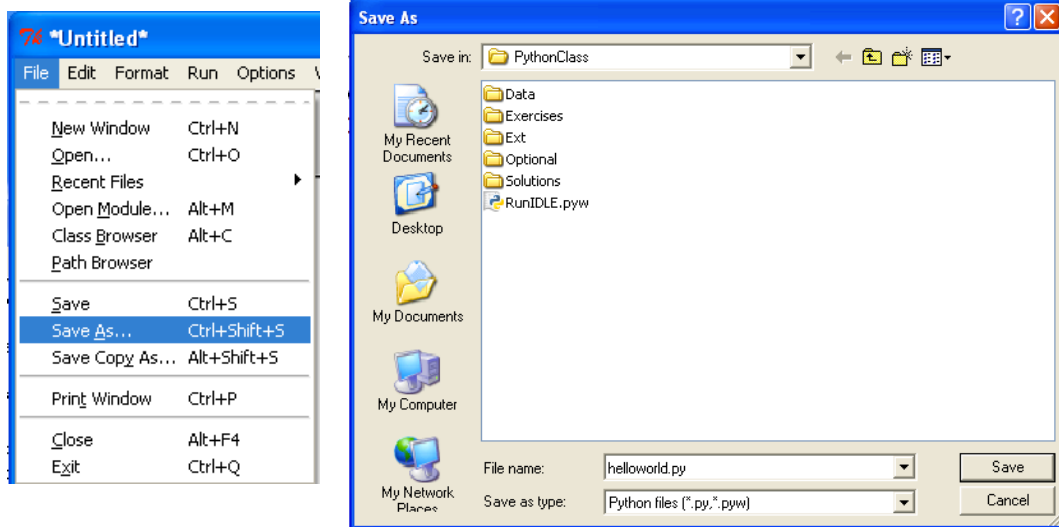
Creating Programs

- Editing a new program in IDLE



Creating Programs

- Saving a new Program in IDLE

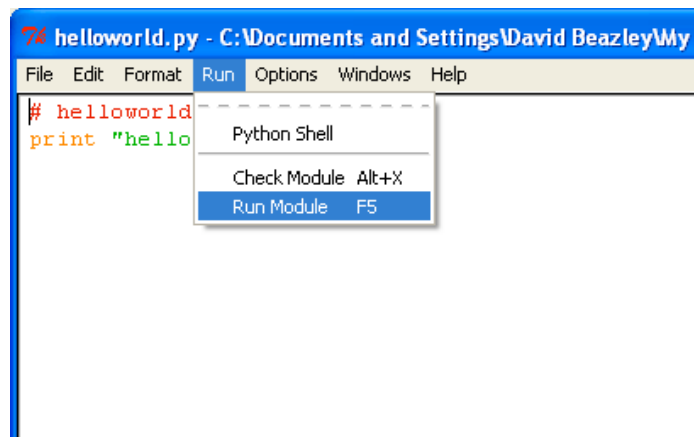


Copyright (C) 2014, <http://www.dabeaz.com>

I-21

Running Programs (IDLE)

- Select "Run Module" (F5)



- Will see output in IDLE shell window

Copyright (C) 2014, <http://www.dabeaz.com>

I-22

Running Programs

- In production environments, Python may be run from command line or a script

- Command line (Unix)

```
bash % python helloworld.py  
hello world  
bash %
```

- Command shell (Windows)

```
C:\SomeFolder>helloworld.py  
hello world
```

```
C:\SomeFolder>c:\python27\python helloworld.py  
hello world
```

A Sample Program

- The Sears Tower Problem

One morning, you go out and place a dollar bill on the sidewalk by the Sears tower. Each day thereafter, you go out double the number of bills. How long does it take for the stack of bills to exceed the height of the tower?



A Sample Program

```
# sears.py

bill_thickness = 0.11 * 0.001    # Meters (0.11 mm)
sears_height    = 442            # Height (meters)
num_bills      = 1
day            = 1

while num_bills * bill_thickness < sears_height:
    print day, num_bills, num_bills * bill_thickness
    day = day + 1
    num_bills = num_bills * 2

print "Number of days", day
print "Number of bills", num_bills
print "Final height", num_bills * bill_thickness
```

A Sample Program

- Output

```
bash % python sears.py
1 1 0.00011
2 2 0.00022
3 4 0.00044
4 8 0.00088
5 16 0.00176
6 32 0.00352
...
21 1048576 115.34336
22 2097152 230.68672
Number of days 23
Number of bills 4194304
Final height 461.37344
```

Exercise 1.2

Time: 10 minutes

Python 101 : Statements

- A Python program is a sequence of statements
- Each statement is terminated by a newline
- Statements are executed one after the other until you reach the end of the file.
- When there are no more statements, the program stops

Python 101 : Comments

- Comments are denoted by #

```
# This is a comment
height = 442           # Meters
```

- Extend to the end of the line
- There are no block comments in Python (e.g., /* ... */).

Python 101:Variables

- A variable is just a name for some value
- Variable names follow same rules as C [A-Za-z_][A-Za-z0-9_]*
- You do not declare types (int, float, etc.)

```
height = 442           # An integer
height = 442.0         # Floating point
height = "Really tall" # A string
```

- Differs from C++/Java where variables have a fixed type that must be declared.

Python 101: Case Sensitivity

- Python is case sensitive
- These are all different variables:

```
name = "Jake"  
Name = "Elwood"  
NAME = "Guido"
```

- Language statements are always lower-case

```
print "Hello World"      # OK  
PRINT "Hello World"     # ERROR  
  
while x < 0:             # OK  
WHILE x < 0:            # ERROR
```

- So, no shouting please...

Python 101: Cleaning up

- Python has garbage collection
- Values are destroyed when no longer used

```
s = "Guido"  
s = 42          # Previous value destroyed
```

- Or you can delete manually

```
del s
```

- Mostly, you don't worry about it (just works)

Python 101: Looping

- The while statement executes a loop

```
while num_bills * bill_thickness < sears_height:  
    print day, num_bills, num_bills * bill_thickness  
    day = days + 1  
    num_bills = num_bills * 2
```

- Executes the indented statements underneath while the condition is true

Python 101 : Indentation

- Indentation used to denote blocks of code
- Indentation must be consistent

```
while num_bills * bill_thickness < sears_height:  
    print day, num_bills, num_bills * bill_thickness  
        day = days + 1  
    num_bills = num_bills * 2
```

(error)

- Colon (:) indicates the start of a block

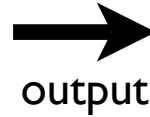
```
while num_bills * bill_thickness < sears_height:
```



Python 101 : Indentation

- Mixing tabs/spaces can explode heads

```
1 #hello.py
2
3 for i in range(10):
4     if i == 5:
5         print 'Hello'
6         print 'World'
7     print 'Done'
8
```



```
World
World
World
World
World
Hello
World
World
World
World
World
World
World
World
World
Done
```

Python tab stops are every 8 characters, but the above editor is showing 4 characters. Second 'print' is indented using spaces and aligns with the first tab.

DON'T USE TABS!

Python 101 : Indentation

- There is a preferred indentation style
 - Always use spaces
 - Use 4 spaces per level
 - Avoid tabs (convert to spaces)
- Always use a Python-aware editor

Python 101 : Conditionals

- If-else

```
if a < b:  
    print "Computer says no"  
else:  
    print "Computer says yes"
```

- If-elif-else

```
if a == '+':  
    op = PLUS  
elif a == '-':  
    op = MINUS  
elif a == '*':  
    op = TIMES  
else:  
    op = UNKNOWN
```

Python 101 : Relations

- Relational operators

< > <= >= == !=

- Boolean expressions (and, or, not)

```
if b >= a and b <= c:  
    print "b is between a and c"  
  
if not (b < a or b > c):  
    print "b is still between a and c"
```

Python 101 : Truth Values

- Evaluates as "True"
 - Non-zero numbers
 - Non-empty strings
 - Non-empty containers (lists, etc.)
- Evaluates as "False"
 - 0 (Zero)
 - Empty strings or containers

```
if x:  
    ... statements ...
```

Python 101 : Printing

- The print statement

```
print x  
print x,y,z  
print "Your name is", name  
print x, # Omits newline
```
- Produces a single line of text
- Items are separated by spaces
- Always prints a newline unless a trailing comma is added after last item

Python 101 : User Input

- To read a line of typed user-input

```
name = raw_input("Enter your name:")
```

- Prints a prompt, returns the typed response
- This might be useful for small programs or for simple debugging
- It is not widely used for real programs (we're rarely going to use it in this class)

Python 101 : pass statement

- Sometimes you will need to specify an empty block of code (like {} in C/Java)

```
if name in namelist:  
    # Not implemented yet (or nothing)  
    pass  
else:  
    statements
```

- pass is a "no-op" statement
- It does nothing, but serves as a placeholder for statements (possibly to be added later)

Python 101 : Long Lines

- Sometimes you get long statements that you want to break across multiple lines
- Use the line continuation character (`\`)

```
if product=="game" and type=="pirate memory" \  
    and age >= 4 and age <= 8:  
    print "I'll take it!"
```

- However, not needed for code in `()`, `[]`, or `{}`

```
if (product=="game" and type=="pirate memory"  
    and age >= 4 and age <= 8):  
    print "I'll take it!"
```

Exercise 1.3

Time: 15 minutes

Basic Datatypes

- Python only has a few primitive types of data
- Numbers
- Strings (character text)

Numbers

- Python has 4 types of numbers
 - Booleans
 - Integers
 - Floating point
 - Complex (imaginary numbers)

Booleans

- Two values: True, False

```
a = True
b = False
```

- Evaluated as integers with value 1,0

```
c = 4 + True    # c = 5
d = False
if d == 0:
    print "d is False"
```

- Although doing that in practice would be odd

Integers

- Signed values of arbitrary size

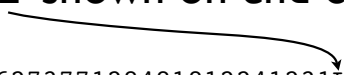
```
a = 37
b = -299392993727716627377128481812241231
c = 0x7fa8          # Hexadecimal
d = 0o253           # Octal
e = 0b10001111     # Binary
```

- There are two internal representations

- int : Small values (less than 32-bits in size)
- long : Large values (arbitrary size)

- Sometimes see 'L' shown on end of large values

```
>>> b
-299392993727716627377128481812241231L
```



Integer Operations

+	Add
-	Subtract
*	Multiply
/	Divide
//	Floor divide
%	Modulo
**	Power
<<	Bit shift left
>>	Bit shift right
&	Bit-wise AND
	Bit-wise OR
^	Bit-wise XOR
~	Bit-wise NOT
abs(x)	Absolute value
pow(x,y[,z])	Power with optional modulo (x**y)%z
divmod(x,y)	Division with remainder

Integer Division

- Classic division (/) - truncates

```
>>> 5/4
1
>>>
```

- Floor division (//) - truncates (same)

```
>>> 5//4
1
>>>
```

- Future division (/) - Converts to float

```
>>> from __future__ import division
>>> 5/4
1.25
```

- In Python 3, / always produces a float
- If truncation is intended, use //

Floating point (float)

- Use a decimal or exponential notation

```
a = 37.45
b = 4e5
c = -1.345e-10
```

- Represented as double precision using the native CPU representation (IEEE 754)

```
17 digits of precision
Exponent from -308 to 308
```

- Same as the C double type

Floating point

- Be aware that floating point numbers are inexact when representing decimal values.

```
>>> a = 2.1 + 4.2
>>> a == 6.3
False
>>> a
6.3000000000000001
>>>
```

- This is not Python, but the underlying floating point hardware on the CPU.
- The result of a calculation may not be quite what you expect (emphasis, not a Python bug)

Floating Point Operators

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo (remainder)
**	Power
pow(x,y [,z])	Power modulo (x**y)%z
abs(x)	Absolute value
divmod(x,y)	Division with remainder

- Additional functions are in the math module

```
import math
a = math.sqrt(x)
b = math.sin(x)
c = math.cos(x)
d = math.tan(x)
e = math.log(x)
```

Converting Numbers

- Type name can be used to convert

```
a = int(x)           # Convert x to integer
b = float(x)        # Convert x to float
```

- Example:

```
>>> a = 3.14159
>>> int(a)
3
>>>
```

- Also work with strings containing numbers

```
>>> a = "3.14159"
>>> float(a)
3.14159
>>>
```

Strings

- Written in programs with quotes

```
a = "Yeah but no but yeah but..."
```

```
b = 'computer says no'
```

```
c = '''  
Look into my eyes, look into my eyes,  
the eyes, the eyes, the eyes,  
not around the eyes,  
don't look around the eyes,  
look into my eyes, you're under.  
'''
```

- Standard escape characters work (e.g., '\n')
- Triple quotes capture all literal text enclosed

String Escape Codes

- In quotes, standard escape codes work

'\n'	Line feed
'\r'	Carriage return
'\t'	Tab
'\xhh'	Hexadecimal value
'\"'	Literal quote
'\\'	Backslash

- The codes are inspired by C

String Representation

- An ordered sequence of bytes (characters)
- Stores 8-bit data (ASCII)
- May contain binary data, control characters, etc.
- Strings are frequently used for both text and for raw-data of any kind

String Representation

- Strings work like an array : `s[n]`

```
a = "Hello world"
b = a[0]          # b = 'H'
c = a[4]          # c = 'o'
d = a[-1]         # d = 'd' (Taken from end of string)
```

- Slicing/substrings : `s[start:end]`

```
d = a[:5]        # d = "Hello"
e = a[6:]        # e = "world"
f = a[3:8]       # f = "lo wo"
g = a[-5:]       # g = "world"
```

- Concatenation (+)

```
a = "Hello" + "World"
b = "Say " + a
```

More String Operations

- Length (len)

```
>>> s = "Hello"  
>>> len(s)  
5  
>>>
```

- Membership test (in, not in)

```
>>> 'e' in s  
True  
>>> 'x' in s  
False  
>>> "hi" not in s  
True
```

- Replication (s*n)

```
>>> s = "Hello"  
>>> s*5  
'HelloHelloHelloHelloHello'  
>>>
```

String Methods

- Strings have "methods" that perform various operations with the string data.

- Stripping any leading/trailing whitespace

```
t = s.strip()
```

- Case conversion

```
t = s.lower()  
t = s.upper()
```

- Replacing text

```
t = s.replace("Hello", "Hallo")
```

More String Methods

```
s.endswith(suffix) # Check if string ends with suffix
s.find(t)          # First occurrence of t in s
s.index(t)        # First occurrence of t in s
s.isalpha()       # Check if characters are alphabetic
s.isdigit()       # Check if characters are numeric
s.islower()       # Check if characters are lower-case
s.isupper()       # Check if characters are upper-case
s.join(slist)     # Joins lists using s as delimiter
s.lower()         # Convert to lower case
s.replace(old,new) # Replace text
s.rfind(t)        # Search for t from end of string
s.rindex(t)       # Search for t from end of string
s.split([delim])  # Split string into list of substrings
s.startswith(prefix) # Check if string starts with prefix
s.strip()         # Strip leading/trailing space
s.upper()         # Convert to upper case
```

- Consult a reference for gory details

String Mutability

- Strings are "immutable" (read only)
- Once created, the value can't be changed

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

- All operations and methods that manipulate string data always create new strings

String Conversions

- Use `str()` to convert a value to a string

```
>>> x = 42
>>> str(x)
'42'
>>>
```

- Resulting text is same as produced by `print`

Special Strings

- String literals are sometimes prefixed by a special code that affect semantics

```
u'Jalape\u00f1o'      # Unicode (international chars)
b'\x12\x27@\x00'     # Bytes
r'c:\newdata\test'   # Raw (uninterpreted backslash)
```

- Unicode strings store multi-byte characters
- Raw strings leave backslashes intact
- 8-bit bytes only allowed in byte strings

Exercise 1.4

Time: 10 minutes

String Splitting

- Strings often represent fields of data
- To work with each field, split into a list

```
>>> line = 'GOOG,100,490.10'  
>>> fields = line.split(',')  
>>> fields  
['GOOG', '100', '490.10']  
>>>
```

- Example: When reading data from a file, you might read each line and then split the line into columns.

Lists

- A array of arbitrary values

```
names = [ 'Elwood', 'Jake', 'Curtis' ]  
nums  = [ 39, 38, 42, 65, 111]
```

- Adding new items (append, insert)

```
names.append('Murphy')    # Adds at end  
names.insert(2,'Aretha') # Inserts in middle
```

- Concatenation : s + t

```
s = [1, 2, 3]  
t = ['a', 'b']  
  
s + t  —————> [1, 2, 3, 'a', 'b']
```

Lists (cont)

- Lists are indexed by integers (starting at 0)

```
names = [ 'Elwood', 'Jake', 'Curtis' ]  
  
names[0] —————> 'Elwood'  
names[1] —————> 'Jake'  
names[2] —————> 'Curtis'
```

- Negative indices are from the end

```
names[-1] —————> 'Curtis'
```

- Changing one of the items

```
names[1] = 'Joliet Jake'
```

More List Operations

- Length (len)

```
>>> names = ['Elwood', 'Jake', 'Curtis']
>>> len(names)
3
>>>
```

- Membership test (in, not in)

```
>>> 'Elwood' in names
True
>>> 'Britney' not in names
True
>>>
```

- Replication (s * n)

```
>>> s = [1, 2, 3]
>>> s * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
```

List Searching

- Finding the first position of an item

```
>>> names = ['Elwood', 'Jake', 'Curtis']
>>> names.index('Curtis')
2
>>> names[2]
'Curtis'
>>>
```

List Removal

- Removing an item

```
names.remove('Curtis')
```

- Deleting an item by index

```
del names[2]
```

- Removal results in items moving down to fill the space vacated (i.e., no "holes").

List Iteration

- Iterating over the list contents

```
for name in names:  
    # use name  
    ...
```

- Similar to a 'foreach' statement from other programming languages

List Sorting

- Lists can be sorted "in-place" (sort method)

```
s = [10, 1, 7, 3]
s.sort()           # s = [1, 3, 7, 10]
```

- Sorting in reverse order

```
s = [10, 1, 7, 3]
s.sort(reverse=True) # s = [10, 7, 3, 1]
```

- Sorting works with any ordered data

```
s = ['foo', 'bar', 'spam']
s.sort()           # s = ['bar', 'foo', 'spam']
```

Lists and Math

- Caution : Lists weren't designed for "math"

```
>>> nums = [1, 2, 3, 4, 5]
>>> nums * 2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> nums + [10, 11, 12, 13, 14]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
>>>
```

- They don't represent vectors/matrices
- Not the same as in MATLAB, Octave, IDL, etc.
- There are some add-ons for this (e.g., numpy)

Exercise 1.5

Time: 10 minutes

File Input and Output

- Opening a file

```
f = open('foo.txt', 'r')    # Open for reading
g = open('bar.txt', 'w')    # Open for writing
```

- To read data

```
data = f.read([maxbytes])  # Read up to maxbytes bytes
```

- To write text to a file

```
g.write('some text')
```

- To close when you're done

```
f.close()
```

Reading File Data

- Reading an entire file all at once as a string

```
f = open(filename, 'r')
data = f.read()
f.close()
```

- Reading an entire text-file line-by-line

```
f = open(filename, 'r')
for line in f:
    # Process the line
    ...
f.close()
```

Reading File Data

- End-of-file indicated by an empty string

```
data = f.read(nbytes)
if data == '':
    # No data read. EOF
    ...
```

- Example: Reading a file in fixed-size chunks

```
f = open(filename, 'r')
while True:
    chunk = f.read(chunksize)
    if chunk == '':
        break
    # Process the chunk
    ...
f.close()
```

Writing To a File

- Writing string data

```
f = open('outfile', 'w')
f.write('Hello World\n')
...
f.close()
```

- Redirecting the print statement

```
f = open('outfile', 'w')
print >>f, 'Hello World'
...
f.close()
```

File Management

- Files should be properly closed when done

```
f = open(filename, 'r')
# Use the file f
...
f.close()
```

- In modern Python (2.6 or newer), use "with"

```
with open(filename, 'r') as f:
    # Use the file f
    ...
```

file f → statements
closed here

- This automatically closes the file when control leaves the indented code block

Exercise 1.6

Time: 15 minutes

Type Conversion

- In Python, you must be careful about converting data to an appropriate type

```
x = '37'      # Strings
y = '42'
z = x + y     # z = '3742' (concatenation)

x = 37
y = 42
z = x + y     # z = 79 (integer +)
```

- Differs from Perl/PHP where "+" is assumed to be numeric arithmetic (even on strings)

```
$x = '37';
$y = '42';
$z = $x + $y; # $z = 79
```


Simple Functions

- Use functions for code you want to reuse

```
def sumcount(n):  
    '''Returns the sum of the first n integers'''  
    total = 0  
    while n > 0:  
        total += n  
        n -= 1  
    return total
```

- Calling a function

```
a = sumcount(100)
```

- A function is just a series of statements that perform some task and return a result

Library Functions

- Python comes with a large standard library
- Library modules accessed using import

```
import math  
x = math.sqrt(10)  
  
import urllib  
u = urllib.urlopen('http://www.python.org/index.html')  
data = u.read()
```

- Will cover in more detail later

Exception Handling

- Errors are reported as exceptions
- An exception causes the program to stop

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

- For debugging, message describes what happened, where the error occurred, along with a traceback.

Exceptions

- Exceptions can be caught and handled
- To catch, use try-except statement

```
for line in f:
    fields = line.split()
    try:
        shares = int(fields[1])
    except ValueError:
        print "Couldn't parse", line
    ..
```

Name must match the kind of error
you're trying to catch

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

Exceptions

- To raise an exception, use the raise statement

```
raise RuntimeError("What a kerfuffle")
```

- Will cause the program to abort with an exception traceback (unless caught by try-except)

```
% python foo.py  
Traceback (most recent call last):  
  File "foo.py", line 21, in <module>  
    raise RuntimeError("What a kerfuffle")  
RuntimeError: What a kerfuffle
```

Summary

- This has been an overview of simple Python
- Enough to write basic programs
- Just have to know the core datatypes and a few basics (loops, conditions, etc.)

Exercise 1.7

Time: 15 minutes

Section 2

Working with Data

Overview

- Most programs work with data
- In this section, we look at how Python programmers represent and work with data
- Common programming idioms
- How to (not) shoot yourself in the foot

Primitive Datatypes

- Python has a few primitive types of data
 - Integers
 - Floating point numbers
 - Strings (text)
- Obviously, all programs use these

None type

- Nothing, nil, null, nada

```
logfile = None
```

- This is often used as a placeholder for optional or missing value

```
if logfile:  
    logfile.write('Some message')
```

- If you don't assign logfile to something, the above code would crash (undefined variable)

Data Structures

- Real programs have more complex data
- Example: A holding of stock

```
100 shares of GOOG at $490.10
```

- An "object" with three parts
 - Name ("GOOG", a string)
 - Number of shares (100, an integer)
 - Price (490.10, a float)

Tuples

- A collection of values grouped together
- Example:

```
s = ('GOOG', 100, 490.1)
```

- Sometimes the () are omitted in syntax

```
s = 'GOOG', 100, 490.1
```

- Special cases (0-tuple, 1-tuple)

```
t = ()          # An empty tuple  
w = ('GOOG',)  # A 1-item tuple
```

Tuple Use

- Tuples are usually used to represent simple records and data structures

```
contact = ('David Beazley', 'dave@dabeaz.com')
stock   = ('GOOG', 100, 490.1)
host    = ('www.python.org', 80)
```

- Basically, a single "object" of multiple parts
- Analogy: A single row in a database table

Tuples (cont)

- Tuple contents are ordered (like an array)

```
s = ('GOOG', 100, 490.1)
name  = s[0]      # 'GOOG'
shares = s[1]    # 100
price  = s[2]    # 490.1
```

- However, the contents can't be modified

```
>>> s[1] = 75
TypeError: object does not support item assignment
```

- You can, however, make a new tuple

```
s = (s[0], 75, s[2])
```


Tuple Packing

- Tuples are focused on packing and unpacking data, not storing distinct items in a list
- Packing multiple values into a tuple

```
s = ('GOOG', 100, 490.1)
```

- The tuple is then easy to pass around to other parts of a program as a single object

Tuple Unpacking

- To use the tuple elsewhere, you typically unpack its parts into variables
- Unpacking values from a tuple

```
(name, shares, price) = s  
print "Cost", shares*price
```

- Note: The () syntax is sometimes omitted

```
name, shares, price = s
```

Tuple Commentary

- Are tuples just a read-only list? No.
- Tuples are most often used for a single record consisting of multiple parts

```
record = ('GOOG', 100, 490.1)
```

- Lists are usually a collection of distinct items (typically all of the same type)

```
names = ['Elwood', 'Jake', 'Curtis']
```

Dictionaries

- A hash table or associative array
- A collection of values indexed by "keys"
- The keys serve as field names
- Example:

```
s = {  
    'name' : 'GOOG',  
    'shares' : 100,  
    'price' : 490.1  
}
```

Dictionaries

- Getting values: Just use the key names

```
>>> print s['name'],s['shares']
GOOG 100
>>> s['price']
490.10
>>>
```

- Adding/modifying values :Assign to key names

```
>>> s['shares'] = 75
>>> s['date'] = '6/6/2007'
>>>
```

- Deleting a value

```
>>> del s['date']
>>>
```

Dictionaries

- Dictionaries are useful when
 - there are many different values
 - The values will be modified/manipulated
- You also get better code clarity

`s['date']` VS `s[4]`

Exercise 2.1

Time : 10 minutes

Containers

- Programs often have to work many objects
 - A portfolio of stocks
 - A table of stock prices
- Three choices:
 - Lists (ordered data)
 - Dictionaries (unordered data)
 - Sets (unordered collection)

Lists as a Container

- Use a list when the order of data matters
- Lists can hold any kind of object
- Example: A list of tuples

```
portfolio = [  
    ('GOOG', 100, 490.1),  
    ('IBM', 50, 91.3),  
    ('CAT', 150, 83.44)  
]  
  
portfolio[0] → ('GOOG', 100, 490.1)  
portfolio[1] → ('IBM', 50, 91.3)
```

List Construction

- Example of building a list from scratch

```
records = []          # Initial empty list  
  
# Use .append() to add more items  
records.append(('GOOG', 100, 490.10))  
records.append(('IBM', 50, 91.3))  
...
```

- Example: Reading records from a file

```
records = []          # Initial empty list  
  
f = open('portfolio.csv', 'r')  
for line in f:  
    row = line.split(',')  
    records.append((row[0], int(row[1]), float(row[2])))
```

Dicts as a Container

- Dictionaries are useful if you want fast random lookups (by key name)
- Example: A dictionary of stock prices

```
prices = {
    'GOOG' : 513.25,
    'CAT'   : 87.22,
    'IBM'   : 93.37,
    'MSFT'  : 44.12
    ...
}

>>> prices['IBM']
93.37
>>> prices['GOOG']
513.25
>>>
```

Copyright (C) 2014, <http://www.dabeaz.com>

2-19

Dict Construction

- Example of building a dict from scratch

```
prices = {}      # Initial empty dict

# Insert new items
prices['GOOG'] = 513.25
prices['CAT'] = 87.22
prices['IBM'] = 93.37
```

- Example: Populating from a file

```
prices = {}      # Initial empty dict

f = open('prices.csv', 'r')
for line in f:
    row = line.split(',')
    prices[row[0]] = float(row[1])
```

Copyright (C) 2014, <http://www.dabeaz.com>

2-20

Dictionary Lookups

- To test for existence of a key

```
if key in d:  
    # Yes  
else:  
    # No
```

- Looking up a value that might not exist

```
name = d.get(key, default)
```

- Example:

```
>>> prices.get('IBM', 0.0)  
93.37  
>>> prices.get('SCOX', 0.0)  
0.0  
>>>
```

Sets

- Sets

```
tech_stocks = set(['IBM', 'AAPL', 'MSFT'])
```

- Holds collection of unordered items
- No duplicates, support common set ops

```
>>> stocks = set(['AA', 'MSFT', 'GE', 'CAT'])  
>>> stocks | tech_stocks      # Union  
set(['AA', 'GE', 'IBM', 'AAPL', 'MSFT', 'CAT'])  
>>> stocks & tech_stocks     # Intersection  
set(['MSFT'])  
>>> stocks - tech_stocks     # Difference  
set(['AA', 'GE', 'CAT'])  
>>>
```

- Useful for membership tests

Set Example

- Example: Eliminating duplicates from a list

```
names = ['IBM', 'AAPL', 'GOOG', 'IBM', 'GOOG', 'YHOO']  
  
# Eliminate duplicates while maintaining original order  
seen = set()  
unique_names = []  
  
for name in names:  
    if name not in seen:  
        unique_names.append(name)  
        seen.add(name)  
  
print unique_names    # ['IBM', 'AAPL', 'GOOG', 'YHOO']
```

Exercise 2.2

Time : 20 minutes

Formatted Output

- When working with data, you often want to produce structured output (tables, etc.).

Name	Shares	Price
AA	100	32.20
IBM	50	91.10
CAT	150	83.44
MSFT	200	51.23
GE	95	40.37
MSFT	50	65.10
IBM	100	70.44

String Formatting

- Formatting operator (%)

```
>>> 'The value is %d' % 3
'The value is 3'
>>> '%5d %-5d %10d' % (3,4,5)
'   3 4                5'
>>> '%0.2f' % (3.1415926,)
'3.14'
```

- Requires single item or a tuple on right

- Commonly used with print

```
print '%d %0.2f %s' % (index,val,label)
```

- Format codes are same as with C printf()

Format Codes

<code>%d</code>	Decimal integer
<code>%u</code>	Unsigned integer
<code>%x</code>	Hexadecimal integer
<code>%f</code>	Float as [-]m.dddddd
<code>%e</code>	Float as [-]m.dddddde+-xx
<code>%g</code>	Float, but selective use of E notation
<code>%s</code>	String
<code>%c</code>	Character
<code>%%</code>	Literal %
<code>%10d</code>	Decimal in a 10-character field (right align)
<code>%-10d</code>	Decimal in a 10-character field (left align)
<code>%0.2f</code>	Float with 2 digit precision
<code>%40s</code>	String in a 40-character field (right align)
<code>%-40s</code>	String in a 40-character field (left align)

format() Function

- Formatting of single values

```
>>> x = 12.3456
>>> format(x, "0.2f")
'12.35'
>>> format(x, ">10.2f")      # Right justify (>)
'      12.35'
>>> format(x, "<10.2f")      # Left justify (<)
'12.35 '
>>> format(x, "^10.2f")      # Centering (^)
'  12.35  '
>>> format(x, "=^10.2f")     # Fill character (=)
'==12.35==='
>>>
```

- Note: format codes similar to % operator

format() method

- .format() method of strings

```
>>> 'The value is {0}'.format(3)
'The value is 3'
>>> '{0:5d} {1:<5d} {2:10d}'.format(3,4,5)
'   3 4           5'
>>> '{name} has {n} messages'.format(name='Dave', n=37)
'Dave has 37 messages'
>>>
```

- {n} placeholder replaced by argument n
- {keyword} replace by keyword argument

Exercise 2.3

Time : 20 minutes

Working with Sequences

- Python has three "sequence" datatypes

```
a = 'Hello'           # String
b = [1, 4, 5]         # List
c = ('GOOG', 100, 490.1) # Tuple
```

- Sequences are ordered : $s[n]$

```
a[0]  —————> 'H'
b[-1] —————> 5
c[1]  —————> 100
```

- Sequences have a length : $\text{len}(s)$

```
len(a) —————> 5
len(b) —————> 3
len(c) —————> 3
```

Working with Sequences

- Sequences can be replicated : $s * n$

```
>>> a = 'Hello'
>>> a * 3
'HelloHelloHello'
>>> b = [1, 2, 3]
>>> b * 2
[1, 2, 3, 1, 2, 3]
>>>
```

- Similar sequences can be concatenated : $s + t$

```
>>> a = (1, 2, 3)
>>> b = (4, 5)
>>> a + b
(1, 2, 3, 4, 5)
>>>
```

Sequence Slicing

- Slicing operator : $s[start:end]$

$a = [0, 1, 2, 3, 4, 5, 6, 7, 8]$

$a[2:5] \longrightarrow [2, 3, 4]$

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

$a[-5:] \longrightarrow [4, 5, 6, 7, 8]$

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

$a[:3] \longrightarrow [0, 1, 2]$

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

- Indices must be integers
- Slices do not include end value
- If indices are omitted, they default to the beginning or end of the list.

Extended Slices

- Extended slicing: $s[start:end:step]$

$a = [0, 1, 2, 3, 4, 5, 6, 7, 8]$

$a[0:5:2] \longrightarrow [0, 2, 4]$

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

$a[::-2] \longrightarrow [8, 6, 4, 2, 0]$

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

$a[6:2:-1] \longrightarrow [6, 5, 4, 3]$

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

- *step* indicates stride and direction
- *end* index is not included in result
- Go easy on it for code clarity

Sequence Reductions

- `sum(s)`

```
>>> s = [1, 2, 3, 4]
>>> sum(s)
10
>>>
```

- `min(s), max(s)`

```
>>> min(s)
1
>>> max(s)
4
>>> max(t)
'World'
>>>
```

Iterating over a Sequence

- The for-loop iterates over sequence data

```
>>> s = [1, 4, 9, 16]
>>> for i in s:
...     print i
...
1
4
9
16
>>>
```

- On each iteration of the loop, you get new item of data to work with.

Iteration Variables

- Each time through the loop, a new value is placed into an iteration variable

```
for x in s:  
    statements:
```

iteration variable

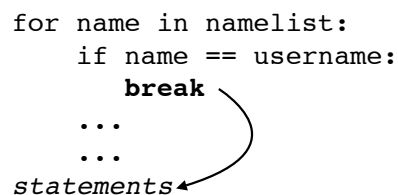


- Overwrites the previous value (if any)
- After the loop finishes, the variable retains the last value

break statement

- Breaking out of a loop (exiting)

```
for name in namelist:  
    if name == username:  
        break  
    ...  
    ...  
statements
```



- Only applies to the inner-most loop

continue statement

- Skipping to the next iteration

```
for line in lines:
    if line == '':
        continue
    # More statements
    ...
```

(An arrow in the original image points from the `continue` statement to the `for` loop header.)

Skip empty lines

- Useful if the current item isn't of interest or needs to be ignored in processing

Looping over integers

- If you simply need to count, use `xrange()`
- `xrange([start,] end [,step])`

```
for i in xrange(100):
    # i = 0,1,...,99

for j in xrange(10,20):
    # j = 10,11,..., 19

for k in xrange(10,50,2):
    # k = 10,12,...,48
```

- Note: The ending value is never included (this mirrors the behavior of slices)

Caution with range()

- range([start,] end [,step])

```
x = range(100)           # x = [0, 1, ..., 99]
y = range(10, 20)        # y = [10, 11, ..., 19]
z = range(10, 50, 2)     # z = [10, 12, ..., 48]
```

- range() creates a list of integers
- Avoid this code (use xrange() instead)

```
for i in range(N):      # Inefficient!
    statements
```

- xrange() computes values as needed instead

enumerate() Function

- enumerate(sequence [, start = 0])
- Provides a loop counter value

```
names = ['Elwood', 'Jake', 'Curtis']
for i,name in enumerate(names):
    # Loops with i = 0, name = 'Elwood'
    #             i = 1, name = 'Jake'
    #             i = 2, name = 'Curtis'
    ...
```

- Example: Keeping a line number

```
f = open(filename)
for lineno, line in enumerate(f, 1):
    ...
```

enumerate() Function

- enumerate() is a nice shortcut

```
for i,x in enumerate(s):  
    statements
```

- Compare to:

```
i = 0  
for x in s:  
    statements  
    i += 1
```

- Less typing and enumerate() runs slightly faster

for and tuples

- Looping with multiple iteration variables

```
points = [  
    (1, 4), (10, 40), (23, 14), (5, 6), (7, 8)  
]  
for x,y in points:  
    # Loops with x = 1, y = 4  
    #           x = 10, y = 40  
    #           x = 23, y = 14  
    #           ...
```

tuples
are expanded

- Here, each tuple is unpacked into a set of iteration variables.

zip() Function

- Combines multiple sequences into tuples

```
columns = ['name', 'shares', 'price']  
values  = ['GOOG', 100, 490.1 ]
```

```
pairs = zip(a,b) # [ ('name', 'GOOG'), ('shares', 100),  
                  # ('price', 490.1) ]
```

- One use, looping over over two sequences

```
for name, value in zip(columns,values):  
    ...
```

- Another use: making dictionaries

```
d = dict(zip(columns,values))
```

Exercise 2.4

Time : 15 minutes

List Comprehensions

- Creates a new list by applying an operation to each element of a sequence.

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8, 10]
>>>
```

- Another example:

```
>>> names = ['Elwood', 'Jake']
>>> a = [name.lower() for name in names]
>>> a
['elwood', 'jake']
>>>
```

List Comprehensions

- A list comprehension can also filter

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0]
>>> b
[2, 8, 4, 20]
>>>
```

- Another example

```
>>> f = open('stockreport', 'r')
>>> goog = [line for line in f if 'GOOG' in line]
>>>
```

List Comprehensions

- General syntax

```
[expression for names in sequence if condition]
```

- What it means

```
result = []  
for names in sequence:  
    if condition:  
        result.append(expression)
```

- Can be used anywhere a sequence is expected

```
>>> a = [1, 2, 3, 4]  
>>> sum([x*x for x in a])  
30  
>>>
```

List Comp: Examples

- List comprehensions are hugely useful
- Collecting the values of a specific field

```
stocknames = [s['name'] for s in stocks]
```

- Performing database-like queries

```
a = [s for s in stocks if s['price'] > 100  
    and s['shares'] > 50 ]
```

- Data reductions over sequences

```
cost = sum([s['shares']*s['price'] for s in stocks])
```

Historical Digression

- List comprehensions come from math

```
a = [x*x for x in s if x > 0]    # Python
```

```
a = { x2 | x ∈ s, x > 0 }      # Math
```

- Implemented in several other languages
- But most Python programmers would probably just view this as a "cool shortcut"

List Comp. and Awk

- For Unix hackers, there is a certain similarity between list comprehensions and short one-line awk commands

```
# A Python List Comprehension
totalcost = sum([shares*price
                 for name, shares, price in portfolio])
```

```
# A Unix awk command
totalcost = `awk '{ total += $2*$3 } END { print total }'
             portfolio.dat`
```

- Applying an operation to every line of a file

Exercise 2.5

Time : 15 Minutes

More details on objects

- So far: a tour of the most common types
- Have skipped some critical details
- Memory management
- Copying
- Type checking

The Issue with Assignment

- Many operations in Python are related to "assigning" or "storing" values

```
a = value           # Assignment to a variable
s[n] = value        # Assignment to an list
s.append(value)     # Appending to a list
d['key'] = value    # Adding to a dictionary
```

- A caution : assignment operations never make a copy of the value being assigned
- All assignments are merely reference copies (or pointer copies if you prefer)

Copyright (C) 2014, <http://www.dabeaz.com>

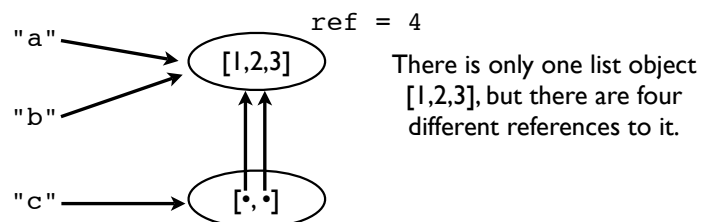
2-55

Assignment Example

- Consider this code fragment:

```
a = [1,2,3]
b = a
c = [a,b]
```

- A picture of the underlying memory



Copyright (C) 2014, <http://www.dabeaz.com>

2-56

Assignment Example

- Modifying a value affects all references

```
>>> a.append(999)
>>> a
[1,2,3,999]
>>> b
[1,2,3,999]
>>> c
[[1,2,3,999], [1,2,3,999]]
>>>
```

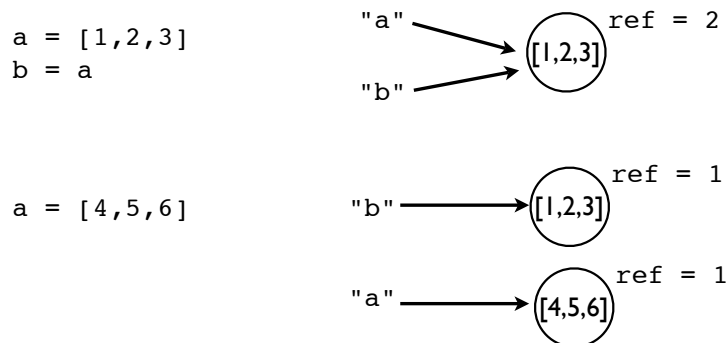
- Notice how a change to the original list shows up everywhere else (yikes!)
- This is because no copies were ever made-- everything is pointing at the same thing

Copyright (C) 2014, <http://www.dabeaz.com>

2-57

Reassigning Values

- Reassigning a value never overwrites the memory used by the previous value



- Variables are names, not memory locations

Copyright (C) 2014, <http://www.dabeaz.com>

2-58

Some Dangers

- If you don't know about this sharing, you will shoot yourself in the foot at some point
- Typical scenario :You modify some data thinking that it's your own private copy and it accidentally corrupts some data in some other part of the program
- Comment:This is one of the reasons why the primitive data types (int, float, string) are immutable (read-only)

Identity and References

- Use the "is" operator to check if two values are exactly the same in memory
- ```
>>> a = [1,2,3]
>>> b = a
>>> a is b
True
>>>
```
- It just compares the object identity (an integer)

```
>>> id(a)
3588944
>>> id(b)
3588944
>>>
```

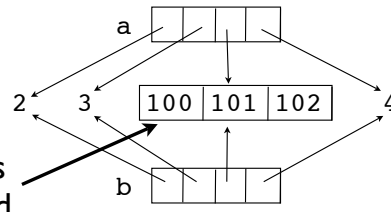
# Shallow Copies

- Lists and dicts have methods for copying

```
>>> a = [2,3,[100,101],4]
>>> b = list(a) # Make a copy
>>> a is b
False
```

- It's a new list, but the list items are shared

```
>>> a[2].append(102)
>>> b[2]
[100,101,102]
>>>
```



- Known as a "shallow copy"

# Deep Copying

- Sometimes you need to make a copy of an object and all objects contained within it

- Use the copy module

```
>>> a = [2,3,[100,101],4]
>>> import copy
>>> b = copy.deepcopy(a)
>>> a[2].append(102)
>>> b[2]
[100,101]
>>>
```

- This is the only safe way to copy something

# Names, Values, Types

- Names do not have a "type"--it's just a name
- However, values do have an underlying type

```
>>> a = 42
>>> b = "Hello World"
>>> type(a)
<type 'int'>
>>> type(b)
<type 'str'>
```

- `type()` will tell you what it is
- The type name is usually a function that creates or converts a value to that type

# Type Checking

- How to tell if an object is a specific type

```
if isinstance(a, list):
 print "a is a list"
```

- Checking for one of many types

```
if isinstance(a, (list, tuple)):
 print "a is a list or tuple"
```

- Caution: Don't go overboard with type checking (can lead to excessive complexity)

# Everything is an object

- Numbers, strings, lists, functions, exceptions, classes, instances, etc...
- All objects are said to be "first-class"
- Meaning: All objects that can be named can be passed around as data, placed in containers, etc., without any restrictions.
- There are no "special" kinds of objects

# First Class Objects

- A simple example:

```
>>> import math
>>> items = [abs, math, ValueError]
>>> items
[<built-in function abs>,
 <module 'math' (builtin)>,
 <type 'exceptions.ValueError'>]
>>> items[0](-45)
45
>>> items[1].sqrt(2)
1.4142135623730951
>>> try:
 x = int("not a number")
except items[2]:
 print "Failed!"

Failed!
>>>
```

A list containing a function, a module, and an exception.

You can use items in the list in place of the original names

# Summary

- Have looked at basic principles of working with data in Python programs
- Brief look at part of the object-model
- A big part of understanding most Python programs.

# Exercise 2.6

Time : 15 Minutes

## Section 3

# Program Organization and Functions

## Overview

- How to organize larger programs
- More details on program execution
- Defining and working with functions
- Exceptions and Error Handling

# Observation

- A large number of Python programmers spend most of their time writing short "scripts"
- One-off problems, prototyping, testing, etc.
- Python is good at this!
- And it what draws many users to Python

# What is a "Script?"

- A "script" is a program that simply runs a series of statements and stops

```
program.py

statement1
statement2
statement3
...
```

- We've been writing scripts to this point



# Problem

- If you write a useful script, it will grow features
- You may apply it to other related problems
- Over time, it might become a critical application
- And it might turn into a huge tangled mess
- So, let's get organized...

# Defining Things

- You must always define things before they get used later on in a program.

```
def square(x):
 return x*x
```

```
a = 42
b = a + 2 # Requires that a is already defined
```

```
z = square(b) # Requires square to be defined
```

- The order is important
- You almost always put the definitions of variables and functions near the beginning

# Defining Functions

- It is a good idea to put all of the code related to a single "task" all in one place

```
def read_prices(filename):
 prices = {}
 f = open(filename)
 f_csv = csv.reader(f)
 for row in f_csv:
 prices[row[0]] = float(row[1])
 f.close()
 return prices
```

- A function also simplifies repeated operations

```
oldprices = read_prices('oldprices.csv')
newprices = read_prices('newprices.csv')
```

# What is a function?

- A function is a sequence of statements

```
def funcname(args):
 statement
 statement
 ...
 return result
```

- Any Python statement can be used inside

```
def foo():
 import math
 print math.sqrt(2)
 help(math)
```

- There are no "special" statements in Python

# Function Definitions

- Functions can be defined in any order

```
def foo(x):
 bar(x)
def bar(x):
 statements
```

```
def bar(x):
 statements
def foo(x):
 bar(x)
```

- Functions must only be defined before they are actually used during program execution

```
foo(3) # foo must be defined already
```

- Stylistically, it is more common to see functions defined in a "bottom-up" fashion

# Bottom-up Style

- Functions are treated as building blocks
- The smaller/simpler blocks go first

```
myprogram.py
def foo(x):
 ...
def bar(x):
 ...
 foo(x)
 ...
def spam(x):
 ...
 bar(x)
 ...
spam(42) # Call spam() to do something
```

Later functions build upon earlier functions

Code that uses the functions appears at the end

# Function Arguments

- Functions operate on passed arguments

```
def square(x):
 return x*x
```

argument

- Argument variables receive their values when the function is called

```
a = square(3)
```

- The argument names are only visible inside the function body (are local to function)

# Function Design

- Try to make functions that only operate on their inputs and which return a proper result

Yes

```
def read_prices(filename):
 prices = {}
 f = open(filename)
 ...
 return prices

prices = read_prices('prices.csv')
```

No

```
filename = 'prices.csv'
prices = {}

def read_prices():
 f = open(filename)
 ...
 return
```

- Depending on external variables makes everything worse. Don't do that.

# Exercise 3.1

Time : 15 minutes

## Default Arguments

- Sometimes you want an optional argument

```
def read_prices(filename, debug=False):
 ...
```

- If a default value is assigned, the argument is optional in function calls

```
d = read_prices('prices.csv')
e = read_prices('prices.dat', True)
```

- Note :Arguments with defaults must appear at the end of the argument list (all non-optional arguments go first)

# Calling a Function

- Consider a simple function

```
def read_prices(filename, debug):
 ...
```

- Calling with "positional" args

```
prices = read_prices('prices.csv', True)
```

- Calling with "keyword" arguments

```
prices = read_prices(filename='prices.csv',
 debug=True)
```

# Keyword Arguments

- Keyword arguments are useful for functions that have optional features/flags

```
def parse_data(data, debug=False, ignore_errors=False):
 ...
```

- Compare and contrast

```
parse_data(data, False, True) # ?????

parse_data(data, ignore_errors=True)
parse_data(data, debug=True)
parse_data(data, debug=True, ignore_errors=True)
```

- Keyword arguments improve code clarity

# Design Tip

- Always give short, but meaningful names to function arguments
- Someone using a function may want to use the keyword calling style

```
d = read_prices('prices.csv', debug=True)
```

- Python development tools will show the names in help features and documentation

```
read_prices(
 (filename, debug=False)
```

# Return Values

- return statement returns a value

```
def square(x):
 return x*x
```

- If no return value, None is returned

```
def bar(x):
 statements
 return
```

```
a = bar(4) # a = None
```

# Multiple Return Values

- A function may return multiple values by returning a tuple

```
def divide(a,b):
 q = a // b # Quotient
 r = a % b # Remainder
 return q, r # Return a tuple
```

- Usage example:

```
x, y = divide(37, 5) # x = 7, y = 2

x = divide(37, 5) # x = (7, 2)
```

# Understanding Variables

- Programs assign values to variables

```
x = value # Global variable

def foo():
 y = value # Local variable
```

- Variable assignments occur outside and inside function definitions
- Variables defined outside are "global"
- Variables inside a function are "local"



# Local Variables

- Variables inside functions are private

```
def read_portfolio(filename):
 portfolio = []
 for line in open(filename):
 fields = line.split()
 s = (fields[0],int(fields[1]),float(fields[2]))
 portfolio.append(s)
 return portfolio
```

- Values not retained or accessible after return

```
>>> stocks = read_portfolio("stocks.dat")
>>> fields
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
NameError: name 'fields' is not defined
>>>
```

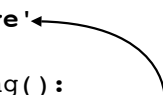
- Don't conflict with variables found elsewhere

# Global Variables

- Functions can access the values of globals

```
name = 'Dave'


def greeting():
 print 'Hello', name
```



- This includes functions

```
def foo():
 ...

def bar():
 foo()
```



# Modifying Globals

- One quirk: Functions can't modify globals

```
x = 0

def foo():
 x = 42
```

- Example:

```
>>> x
0
>>> foo()
>>> x
0 ←————— Notice no change
>>>
```

- All assignments in functions are local

# Modifying Globals

- If you must modify a global variable you must declare it as such

```
x = 0

def foo():
 global x
 x = 42 # Changes the global x above
```

- global declaration must appear before use
- Considered "bad style"
- Avoid entirely if you can (use a class instead)

# Argument Passing

- When you call a function, the argument variables are names for passed values
- If mutable data types are passed (e.g., lists, dicts), they can be modified "in-place"

```
def foo(items):
 items.append(42)

a = [1, 2, 3]
foo(a)
print a # [1, 2, 3, 42]
```

← Modifies the input object

←

- Key point: Function doesn't receive a copy

# Understanding Assignment

- Make sure you understand the subtle difference between modifying a value and reassigning a variable name

- Example:

```
def foo(items):
 items.append(42) # Modifies items list

def bar(items):
 items = [4,5,6] # Binds name 'items' to new list
```

- Reminder :Variable assignment never overwrites memory (the name is simply bound to a new value)

# Exercise 3.2

Time : 30 minutes

## Error Checking

- Python performs no checking or validation of function argument types or values
- A function will work on any data that is compatible with the statements in the function
- Example:

```
def add(x, y):
 return x + y

add(3, 4) # 7
add('Hello', 'World') # 'HelloWorld'
add('3', '4') # '34'
```

# Error Checking

- If there are errors in a function, they will show up at run time (as an exception)

- Example:

```
def add(x, y):
 return x + y

>>> add(3, '4')
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +:
'int' and 'str'
>>>
```

- To verify code, there is a strong emphasis on testing (covered later)

# Exceptions

- Used to signal errors
- Raising an exception (raise)

```
if name not in names:
 raise RuntimeError('Name not found')
```

- Catching an exception (try)

```
try:
 authenticate(username)
except RuntimeError as e:
 print e
```

# Exceptions

- Exceptions propagate to first matching except

```
def foo():
 try:
 bar()
 except RuntimeError as e:
 ...

def bar():
 try:
 spam()
 except RuntimeError as e:
 ...

def spam():
 grok()

def grok():
 ...
 raise RuntimeError('Whoa!')
```

A diagram with two curved arrows. One arrow starts at the `raise RuntimeError('Whoa!')` line in the `grok()` function and points to the `except RuntimeError as e:` line in the `bar()` function. The second arrow starts at the `except RuntimeError as e:` line in the `bar()` function and points to the `except RuntimeError as e:` line in the `foo()` function.

Copyright (C) 2014, <http://www.dabeaz.com>

3-31

# Exceptions

- To handle the exception, statements inside the except block run

```
def bar():
 try:
 grok()
 except RuntimeError as e:
 statements
 statements
 ...

def grok():
 ...
 raise RuntimeError('Whoa!')
```

A diagram with one curved arrow. The arrow starts at the `raise RuntimeError('Whoa!')` line in the `grok()` function and points to the `statements` line in the `except` block of the `bar()` function.

Copyright (C) 2014, <http://www.dabeaz.com>

3-32

# Exceptions

- After handling, execution resumes with the first statement after the try-except

```
def bar():
 try:
 grok()
 except RuntimeError as e:
 statements
 statements
 ...
 ↓ statements
 statements
 ...

def grok():
 ...
 raise RuntimeError("Whoa!")
```

# Builtin-Exceptions

- About two-dozen built-in exceptions

```
ArithmeticError
AssertionError
EnvironmentError
EOFError
ImportError
IndexError
KeyboardInterrupt
KeyError
MemoryError
NameError
ReferenceError
RuntimeError
SyntaxError
SystemError
TypeError
ValueError
```

- Consult reference

# Exception Values

- Most exceptions have an associated value
- More information about what's wrong

```
raise RuntimeError('Invalid user name')
```

- Passed to variable supplied in except

```
try:
 ...
except RuntimeError as e:
 ...
```

- It's an instance of the exception type, but often looks like a string

```
except RuntimeError as e:
 print 'Failed : Reason', e
```

# Catching Multiple Errors

- Can catch different kinds of exceptions

```
try:
 ...
except LookupError as e:
 ...
except RuntimeError as e:
 ...
except IOError as e:
 ...
except KeyboardInterrupt as e:
 ...
```

- Alternatively, if handling is same

```
try:
 ...
except (IOError,LookupError,RuntimeError) as e:
 ...
```



# Catching All Errors

- Catching any exception

```
try:
 ...
except Exception:
 print 'An error occurred'
```

- A really bad idea as shown (don't do it!)

# Exploding Heads

- The wrong way to use exceptions:

```
try:
 go_do_something()
except Exception:
 print 'Computer says no'
```

- This swallows all possible errors
- May make it impossible to debug if code is failing for some reason you didn't expect at all (e.g., uninstalled Python module, etc.)

# A Better Approach

- This is a somewhat more sane approach

```
try:
 go_do_something()
except Exception as e:
 print 'Computer says no. Reason : %s\n' % e
```

- Reports a specific reason for the failure
- It is almost always a good idea to have some mechanism for viewing/reporting errors if you are writing code that catches all possible exceptions

# Reraising an Exception

- Use 'raise' to propagate a caught error

```
try:
 go_do_something()
except Exception as e:
 print 'Computer says no. Reason : %s\n' % e
 raise
```

- Allows you to take action (e.g., logging), but pass the error on to the caller

# Exception Advice

- Don't catch exceptions - fail fast and loud  
(if it's important, someone else will take care of the problem)
- Only catch an exception if you're *that* someone
- That is, only catch errors where you can recover and sanely keep going

# finally statement

- Specifies code that must run regardless of whether or not an exception occurs

```
lock = Lock()
...
lock.acquire()
try:
 ...
finally:
 lock.release() # release the lock
```

- Commonly use to properly manage resources (especially locks, files, etc.)

# with statement

- In modern code, try-finally often replaced with the 'with' statement

```
lock = Lock()
with lock:
 # lock acquired
 ...
lock released

with open(filename) as f:
 # Use the file
 ...
File closed
```

- Defines a usage "context" for a resource
- Only works with certain objects

# Program Exit

- Program exit is handle through exceptions

```
raise SystemExit
raise SystemExit(exitcode)
raise SystemExit('Informative message')
```

- An alternative sometimes seen

```
import sys
sys.exit()
sys.exit(exitcode)
```

- Hard exit (immediate, no cleanup)

```
import os
os._exit(exitcode)
```

# Exercise 3.3

Time : 15 minutes

## Section 4

# Modules and Libraries

## Overview

- How to place code in a module
- Essential standard library modules
- Installing third party libraries

# Modules

- Any Python source file is a module

```
foo.py
def grok(a):
 ...
def spam(b):
 ...
```

- `import` statement loads and executes a module

```
import foo

a = foo.grok(2)
b = foo.spam("Hello")
...
```

# Namespaces

- A module is a collection of named values (i.e., it's said to be a "namespace")
- The names are simply all of the global variables and functions defined in the source file
- After `import`, module name used as a prefix

```
>>> import foo
>>> foo.grok(2)
>>>
```

- Module name is tied to source (foo -> foo.py)

# Global Definitions

- Everything defined in the "global" scope is what populates the module namespace

```
foo.py
x = 42
def grok(a):
 ...

bar.py
x = 37
def spam(a):
 ...
```

These definitions of x  
are different

- Different modules can use the same names and those names don't conflict with each other (modules are isolated)

# Modules as Environments

- Modules form an enclosing environment for all of the code defined inside

```
foo.py
x = 42
def grok(a):
 print(x)
```

global variables are always  
bound to the enclosing  
module (same file)

- Each source file is its own little universe
- This is great!
- What happens in a module stays in a module



# Module Execution

- When a module is imported, all of the statements in the module execute one after another until the end of the file is reached
- The contents of the module namespace are all of the global names that are still defined at the end of the execution process
- If there are scripting statements that carry out tasks in the global scope (printing, creating files, etc.), you will see them run on import

## import as statement

- Changing the name of a module

```
import math as m

def rectangular(r, theta):
 x = r * m.cos(theta)
 y = r * m.sin(theta)
 return x, y
```

- Same as a normal import
- Just a simple renaming in that one file (the one that did the import)

# from module import

- Lifts selected symbols out of a module and makes them available locally

```
from math import sin, cos
```

```
def rectangular(r, theta):
 x = r * cos(theta)
 y = r * sin(theta)
 return x, y
```

- Allows parts of a module to be used without having to type the module prefix
- If library functions are used frequently, this makes them run faster (one less lookup)

# from module import \*

- Takes all symbols from a module and places them into local scope

```
from math import *
```

```
def rectangular(r, theta):
 x = r * cos(theta)
 y = r * sin(theta)
 return x, y
```

- Useful if you are going to use a lot of functions from a module and it's annoying to specify the module prefix all of the time

# from module import \*

- You should almost never use it in practice because it leads to poor code readability
- Example:

```
from math import *
from random import *

...
r = gauss(0.0, 1.0) # In what module?
```

- Makes it very difficult to understand someone else's code if you need to locate the original definition of a library function

## Be Explicit

- In the long run, it's better to be explicit and only import what you actually need

```
from math import sin, cos, sqrt
from random import gauss, uniform

...
r = gauss(0.0, 1.0) # Defined in random (see above)
```

- Of course it depends on the situation
- For interactive sessions and throw-away scripts, "from module import \*" is often preferred (reduces typing and thinking)

# Commentary

- Variations on import do not change the way that modules work

```
import math as m
from math import cos, sin
from math import *
...
```

- import always executes the entire file
- Modules are still isolated environments
- These variations are just manipulating names

# Main Functions

- In many programming languages, there is a concept of a "main" function or method

```
/* C/C++ */
int main(int argc, char *argv[]) {
 ...
}

/* Java */
class myprog {
 public static void main(String args[]) {
 ...
 }
}
```

- It's the first function that executes when an application is launched

# Main Module

- Python has no "main" function or method
- Instead, there is a "main" module
- It's simply the source file that runs first

```
bash % python foo.py
...
```

- Whatever module you give to the interpreter at startup becomes "main"

## \_\_main\_\_ check

- It is standard practice for modules that can run as a main program to use this convention:

```
foo.py
...
if __name__ == '__main__':
 # Running as the main program
 ...
 statements
 ...
```

- Statements enclosed inside the if-statement become the "main" program

# \_\_main\_\_ check

- Important: Any file can either run as main or as a library import

```
bash % python foo.py # Running as main
>>> import foo # Loaded as a module
```

- `__name__` is the name of the module
- As a general rule, you don't want statements that are part of a main program to execute on a library import (hence, the check)

```
if __name__ == '__main__':
 # Does not execute if loaded with import
 ...
```

# Module Loading

- Each module loads and executes once
- Repeated imports just return a reference to the previously loaded module
- `sys.modules` is a dict of all loaded modules

```
>>> import sys
>>> sys.modules.keys()
['copy_reg', '__main__', 'site', '__builtin__',
'encodings', 'encodings.encodings', 'posixpath', ...]
>>>
```

# Locating Modules

- When looking for modules, Python first looks in the same directory as the source file that's executing the import
- If a module can't be found there, an internal module search path is consulted

```
>>> import sys
>>> sys.path
[
 '',
 '/usr/local/lib/python27/python27.zip',
 '/usr/local/lib/python27',
 ...
]
```

# Module Search Path

- `sys.path` contains search path
- Can manually adjust if you need to
- Paths also added via environment variables

```
import sys
sys.path.append("/project/foo/pyfiles")
```

```
% env PYTHONPATH=/project/foo/pyfiles python
Python 2.6.4 (r264:75821M, Oct 27 2009, 19:48:32)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
>>> import sys
>>> sys.path
['', '/project/foo/pyfiles', ...]
```

# Exercise 4.1

Time : 15 Minutes

## Standard Library

- Python includes a large standard library
- Several hundred modules
- System, networking, data formats, etc.
- All accessible via import
- Let's take a tour of the most essential ones



# sys module

- Information related to environment
- Version information
- System limits
- Command line options
- Module search paths
- Standard I/O streams

## sys: Standard I/O

- Standard I/O streams
- By default, print is directed to sys.stdout
- Input read from sys.stdin
- Can redefine or use directly

```
sys.stdout
sys.stderr
sys.stdin
```

```
sys.stdout = open("out.txt", "w")
print >>sys.stderr, "Warning. Unable to connect"
```

# sys: Standard I/O

- Function that allows I/O redirection

```
import sys
def greet(name, outfile=None):
 if outfile is None:
 outfile = sys.stdout
 outfile.write('Hello %s\n' % name)
```

- Example:

```
Write to standard out
greet('Dave')

Write to a file
f = open('somefile.txt', 'w')
greet('Dave', outfile=f)
f.close()
```

# sys: Command Line Opts

- Many programs execute from the shell

```
bash % python report.py portfolio.csv prices.csv
```

- Arguments get placed in sys.argv

```
sys.argv → ['report.py', 'portfolio.csv', 'prices.csv']
```

- Example of processing

```
import sys
if len(sys.argv) != 3:
 raise SystemExit('Usage: %s portfile pricefile' %
 sys.argv[0])
portfile = sys.argv[1]
pricefile = sys.argv[2]
...
```

# Advanced Arguments

- Use argparse for advanced argument parsing

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-o', '--output')
parser.add_argument('-d', '--debug', action='store_true')
parser.add_argument('portfile')
parser.add_argument('pricefile')

Parse the options
args = parser.parse_args()

Retrieve the option values
portfile = args.portfile
pricefile = args.pricefile
outfile = args.output
debug = args.debug
```

# os Module

- Contains operating system functions
- Example: Executing a shell command

```
>>> import os
>>> os.system("mkdir temp")
>>>
```

- Example: Executing a system call

```
>>> os.mkdir("temp")
>>>
```

- Hundreds of other low-level operations

# Environment Variables

- Environment variables (typically set in shell)

```
% setenv NAME dave
% setenv RSH ssh
% python prog.py
```

- `os.environ` dictionary contains values

```
import os
home = os.environ['HOME']
os.environ['HOME'] = '/home/user/guest'
```

- Changes are reflected in Python and any subprocesses created later

# Getting a Directory Listing

- `os.listdir()` function

```
>>> files = os.listdir("/some/path")
>>> files
['foo', 'bar', 'spam']
>>>
```

- `glob` module

```
>>> txtfiles = glob.glob("*.txt")
>>> datfiles = glob.glob("Dat[0-5]*")
>>>
```

- `glob` understands Unix shell wildcards (on all systems)

# os.path Module

- Portable management of path names and files

- Examples:

```
>>> import os.path
>>> os.path.basename("/home/foo/bar.txt")
'bar.txt'
>>> os.path.dirname("/home/foo/bar.txt")
'/home/foo'
>>> os.path.join("home", "foo", "bar.txt")
'home/foo/bar.txt'
>>>
```

- Solves problem of '/' vs. '\' on Unix/Windows

# File Tests

- Testing if a file exists

```
>>> os.path.exists("foo.txt")
True
>>>
```

- Testing if a filename is a regular file

```
>>> os.path.isfile("foo.txt")
True
>>> os.path.isfile("/usr")
False
>>>
```

- Testing if a filename is a directory

```
>>> os.path.isdir("foo.txt")
False
>>> os.path.isdir("/usr")
True
>>>
```

# Pathnames

- Expand into a full absolute path name

```
>>> os.path.abspath("README.html")
'/Users/beazley/Desktop/IntroPython2010/Exercises/
PythonClass/README.html'
>>>
```

- Relative paths (to current working directory)

```
>>> os.path.relpath("/etc/passwd")
'../../../../../../../../etc/passwd'
>>>
```

- Expanding user names

```
>>> os.path.expanduser("~/Desktop")
'/Users/beazley/Desktop'
>>>
```

# File Metadata

- Getting the file size

```
>>> os.path.getsize("foo.txt")
1344L
>>>
```

- Getting the last modification/access time

```
>>> os.path.getmtime("foo.txt")
1175769416.0
>>> os.path.getatime("foo.txt")
1175769491.0
>>>
```

- Note: To decode times, use time module

```
>>> time.ctime(os.path.getmtime("foo.txt"))
'Thu Apr 5 05:36:56 2007'
>>>
```

# Directory Walking

- Walking over a directory tree

```
for path, dirs, files in os.walk(topdir):
 # path = name of current directory
 # dirs = list of all subdirectories in path
 # files = list of all files in path

 # Example: Print out names of .py files
 for filename in files:
 if filename.endswith(".py"):
 print os.path.join(path, filename)
```

- walk() is a little tricky to use at first, but it can be used to carry out operations similar to those performed with Unix 'find'

# Shell Operations (shutil)

- Copying a file

```
>>> shutil.copy("source", "dest")
```

- Moving a file (renaming)

```
>>> shutil.move("old", "new")
```

- Copying a directory tree

```
>>> shutil.copytree("srcdir", "destdir")
```

- Removing a directory tree

```
>>> shutil.rmtree("dir")
```

# time module

- System date and time related functions

```
time.sleep(seconds)
time.clock() # CPU time (seconds)
time.time() # Real time (seconds)
time.localtime([secs]) # Time as a struct
time.ctime([tmstruct]) # Time as a string
```

- Example

```
>>> time.time()
1267128590.589685
>>> time.localtime()
time.struct_time(tm_year=2010, tm_mon=2,
tm_mday=25, tm_hour=14, tm_min=9, tm_sec=54,
tm_wday=3, tm_yday=56, tm_isdst=0)
>>> time.ctime()
'Thu Feb 25 14:09:58 2010'
>>>
```

# datetime module

- A module for more generic representation and manipulation of dates and times

```
>>> from datetime import datetime
>>> cataclysm = datetime(2012,12,21)
>>> cataclysm
datetime.datetime(2012, 12, 21, 0, 0)
>>> today = datetime.today()
>>> d = cataclysm - today
>>> d
datetime.timedelta(1249, 32536, 964510)
>>> d.days
1249
>>>
```

- There are many more features not shown



# subprocess Module

- A module for launching subprocesses
- Capture output of a command as string

```
import subprocess
out = subprocess.check_output(['ls', '-l'])
```

- Low-level subprocess with stdin/stdout pipes

```
p = subprocess.Popen(['wc'],
 stdout=subprocess.PIPE,
 stdin=subprocess.PIPE)
p.stdin.write("Hello World\n")
p.stdin.close()
data = p.stdout.read()
```

## Exercise 4.2

Time : 15 Minutes

# Data Handling

- There are modules for handling various text and binary data processing problems
  - re (regular expressions)
  - xml.etree.ElementTree (XML)
  - json (JSON)
  - struct (binary data)

# Regular Expressions

- Operations involving text patterns
- Extracting text from a document
- Replacing text
- Example: Extracting URLs from text

Go to <http://www.python.org> for more information on Python

# Regex Pattern Syntax

- Regular expression pattern syntax overview

```
foo # Matches the text "foo"
(foo|bar) # Matches the text "foo" or "bar"
(foo)* # Match 0 or more repetitions of foo
(foo)+ # Match 1 or more repetitions of foo
(foo)? # Match 0 or 1 repetitions of foo
(foo){N} # Match N repetitions of foo
[abcde] # Match one of the letters a,b,c,d,e
[a-z] # Match one letter from a,b,...,z
[^a-z] # Match any character except a,b,...,z
. # Match any character except newline
$ # Match end of line
\
* # Match the * character
\
+ # Match the + character
\
d # Match a digit
\
s # Match whitespace
\
w # Match alphanumeric character
```

- Many other advanced options (not shown)

Copyright (C) 2014, <http://www.dabeaz.com>

4-43

# Regex Pattern Examples

- Here are some very simple pattern examples

- A date of the form YYYY/MM/DD

```
(\d+)/(\d+)/(\d+) (Simple)
\d{4}/\d{2}/\d{2} (A bit more precise)
```

- A decimal number (e.g., 12.345, 12.3, 12.)

```
\d+\.\d*
```

- A sequence of hexadecimal digits

```
[0-9A-F]+
```

Copyright (C) 2014, <http://www.dabeaz.com>

4-44

# Writing Regex Patterns

- Patterns are written out as strings
- Usually using raw strings because the `'\'` character has meaning in regex patterns

- Example

```
pat = r'(\d+)/(\d+)/(\d+)'
```

- Recall raw strings don't interpret escapes (`\`)

## re Module

- Support for common text pattern operations

```
>>> import re
>>> text = 'Today is 1/17/2014. Tomorrow is 1/18/2014'

>>> # Find all matches of a pattern
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('1', '17', '2014'), ('1', '18', '2014')]

>>> # Replace a pattern with new text
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2014-1-17. Tomorrow is 2014-1-18'

>>> # Splitting on a pattern
>>> re.split(r'[,:]', '1,2:3,4')
['1', '2', '3', '4']
>>>
```

- Several other operations, but this is basic idea

# re: Groups

- Parenthesized parts of a pattern define groups

```
pat = r'(\d+)/(\d+)/(\d+)'
```

- Groups are assigned numbers

```
pat = r'(\d+)/(\d+)/(\d+)'
 ↑ ↑ ↑
 1 2 3
```

- Numbering is determined by looking at '(' from left to right

# re: Groups

- When searching, groups get separated

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('1', '17', '2014'), ('1', '18', '2014')]
>>>
```

↑            ↑            ↓  
group 1    group 2    group 3

- Use `\N` to refer to groups in replacements

```
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2014-1-17. Tomorrow is 2014-1-18'
>>>
```

# Match Objects

- Certain operations return a 'Match' object

```
>>> m = re.match('r(\d+)/(\d+)/(\d+)', '2014/1/17')
>>> m
<_sre.SRE_Match object at 0x1002cab78>
>>>
```

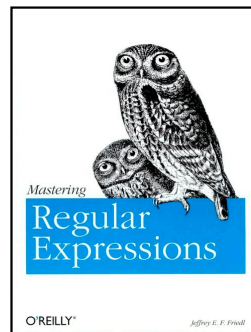
- You can extract matched text from it

```
>>> m.groups()
('2014', '1', '17')
>>> m.group(0)
'2014/1/17'
>>> m.group(1)
'2014'
>>>
```

# re: Comments

- re module is very powerful
- I have only covered the essential basics
- Strongly influenced by Perl
- However, regexs are not an operator
- Reference:

Jeffrey Friedl, "Mastering Regular Expressions", O'Reilly & Associates, 2006.



# Exercise 4.3

(Optional)

Time : 10 Minutes

## XML Parsing

- XML documents use structured markup

```
<contact>
 <name>Elwood Blues</name>
 <address>1060 W Addison</address>
 <city>Chicago</city>
 <zip>60616</zip>
</contact>
```

- Documents made up of elements

```
<name>Elwood Blues</name>
```

- Elements have starting/ending tags

- May contain text and other elements

# XML Example

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe>
 <title>Famous Guacamole</title>
 <description>
 A southwest favorite!
 </description>
 <ingredients>
 <item num="2">Large avocados, chopped</item>
 <item num="1">Tomato, chopped</item>
 <item num="1/2" units="C">White onion, chopped</item>
 <item num="1" units="tbl">Fresh squeezed lemon juice</item>
 <item num="1">Jalapeno pepper, diced</item>
 <item num="1" units="tbl">Fresh cilantro, minced</item>
 <item num="3" units="tsp">Sea Salt</item>
 <item num="6" units="bottles">Ice-cold beer</item>
 </ingredients>
 <directions>
 Combine all ingredients and hand whisk to desired consistency.
 Serve and enjoy with ice-cold beers.
 </directions>
</recipe>
```

# XML Parsing

- XML is a widely used data format
- To parse it, use `xml.etree.ElementTree`
- This is not the only approach, but it is often considered to be the easiest--especially for simple XML problems



# ElementTree Parsing

- Parsing a document

```
from xml.etree.ElementTree import parse
doc = parse('recipe.xml')
```

- Finding one or more elements

```
elem = doc.find('title')
for elem in doc.findall('ingredients/item'):
 statements
```

- Element attributes and properties

```
elem.tag # Element name
elem.text # Element text
elem.get(aname [, default]) # Element attributes
```

# Obtaining Elements

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe>
 <title>Famous Guacamole</title>
 <description>
 A southwest favorite!
 </description>
 <ingredients>
 <item num="2">...
 <item num="1">T...
 <item num="1/2">...
 <item num="1">u...
 <item num="1">J...
 <item num="1">u...
 <item num="3">u...
 <item num="6">u...
 </ingredients>
 <directions>
 Combine all ingredients and hand whisk to desired consistency.
 Serve and enjoy with ice-cold beers.
 </directions>
</recipe>
```

```
doc = parse('recipe.xml')
desc_elem = doc.find('description')
desc_text = desc_elem.text

or

doc = parse('recipe.xml')
desc_text = doc.findtext('description')
```

# Iterating over Elements

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<recipe>
 <title>Famous Guacamole</title>
 <description>
 A southwest favorite!
 </description>
 <ingredients>
 <item num="2">Large avocados, chopped</item>
 <item num="1">Tomato, chopped</item>
 <item num="1/2" units="C">White onion, chopped</item>
 <item num="1" units="tbl">Fresh squeezed lemon juice</item>
 <item num="1">Jalapeno pepper, diced</item>
 <item num="1" units="tbl">Fresh cilantro, minced</item>
 <item num="3" units="tsp">Sea Salt</item>
 <item num="6" units="bottles">Ice-cold beer</item>
 </ingredients>
 <directions>
 Combine all ingredients and hand whisk to desired consistency.
 Serve and enjoy with ice-cold beers.
 </directions>
</recipe>
```

```
doc = parse('recipe.xml')
for item in doc.findall('ingredients/item'):
 statements
```

```
<item num="2">Large avocados, chopped</item>
<item num="1">Tomato, chopped</item>
<item num="1/2" units="C">White onion, chopped</item>
<item num="1" units="tbl">Fresh squeezed lemon juice</item>
<item num="1">Jalapeno pepper, diced</item>
<item num="1" units="tbl">Fresh cilantro, minced</item>
<item num="3" units="tsp">Sea Salt</item>
<item num="6" units="bottles">Ice-cold beer</item>
```

# Element Attributes

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<recipe>
 <title>Famous Guacamole</title>
 <description>
 A southwest favorite!
 </description>
 <ingredients>
 <item num="1" units="tbl">Fresh cilantro, minced</item>
 <item num="3" units="tsp">Sea Salt</item>
 <item num="6" units="bottles">Ice-cold beer</item>
 </ingredients>
 <directions>
 Combine all ingredients and hand whisk to desired consistency.
 Serve and enjoy with ice-cold beers.
 </directions>
</recipe>
```

```
for item in doc.findall('ingredients/item'):
 num = item.get('num')
 units = item.get('units')
```

# JSON Encoding/Decoding

- JSON - JavaScript Object Notation

```
{
 "recipe" : {
 "title" : "Famous Guacomole",
 "description" : "A southwest favorite!",
 "ingredients" : [
 {"num": "2", "item": "Large avocados, chopped"},
 {"num": "1/2", "units": "C", "item": "White onion, chopped"},
 {"num": "1", "units": "tbl", "item": "Fresh squeezed lemon juice"},
 {"num": "1", "item": "Jalapeno pepper, diced"},
 {"num": "1", "units": "tbl", "item": "Fresh cilantro, minced"},
 {"num": "3", "units": "tsp", "item": "Sea Salt"},
 {"num": "6", "units": "bottles", "item": "Ice-cold beer"}
],
 "directions" : "Combine all ingredients and hand whisk to desired consistency. Serve and enjoy with ice-cold beers."
 }
}
```

- Extremely common in web-services

# JSON-Dict Translation

- Converting a dict to JSON

```
>>> import json
>>> s = {'name': 'GOOG', 'shares': 100, 'price': 490.1}
>>> encoded = json.dumps(s)
>>> encoded
'{"price": 490.1, "name": "GOOG", "shares": 100}'
>>>
```

- Converting JSON into a dict

```
>>> json.loads(encoded)
{'price': 490.1, 'name': u'GOOG', 'shares': 100}
>>>
```


# JSON Notes

- JSON is almost identical to Python syntax

```
>>> s = { 'a': True, 'b': None}
>>> json.dumps(s)
'{"a": true, "b": null}'
>>>
```

- Decoded JSON always uses Unicode (UTF-8)

```
>>> json.loads(encoded)
{u'price': 490.1, u'name': u'GOOG', u'shares': 100}
>>>
```



- Unicode is discussed a bit later (Chapter 9)

# Binary Data

- Binary data - low-level machine data
- Examples : 16-bit integers, 32-bit integers, 64-bit double precision floats, packed strings, etc.
- Raw low-level data that you typically encounter with typed programming languages such as C, C++, Java, etc.
- Common with multimedia (images, video) as well as hardware control (serial ports, etc.)

# Binary File I/O

- To obtain binary data, you'll typically read it from a file, pipe, network socket, etc.
- For files, there are special modes

```
f = open(filename,"rb") # Read, binary mode
f = open(filename,"wb") # Write, binary mode
f = open(filename,"ab") # Append, binary mode
```

- Disables all newline translation (reads/writes)
- Required for binary data on Windows
- Optional on Unix (a portability gotcha)

# Binary Data Representation

- To manipulate binary data, use strings
- In Python 2, strings are just byte sequences

```
bytes = '\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00'
\xhh - Encodes an arbitrary byte (hh)
```

- All of the normal string operations work except that you may have to specify a lot of non-text characters using `\xhh` escape codes

# Binary Data Representation

- In Python 2.6 and newer, a special syntax should be used if you are writing byte literal strings in your program

```
header = b'\x89PNG\r\n'
```

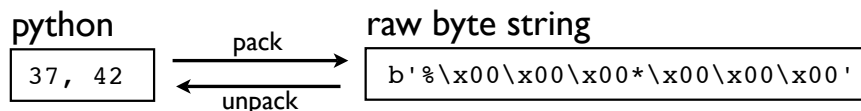


byte string prefix

- This has been added to disambiguate text (Unicode) strings and raw byte strings
- A caution :This is optional in Python 2, but required in Python 3

# Binary Data Packing

- A common operation with binary data is to pack or unpack values from byte strings



- Packing/unpacking is about type conversion
- Converting low-level data to/from built-in Python types such as ints, floats, strings, etc.

# struct module

- Packs/unpacks binary records and structures

```
import struct

Unpack two raw 32-bit integers from a string
x,y = struct.unpack('ii',s)

Pack a set of fields
r = struct.pack('8sif', 'GOOG', 100, 490.10)
```

- Unpacking is used when reading binary data
- Packing is used when writing binary data

# struct module

- Packing/unpacking codes (based on C)

'c'	char (1 byte string)
'b'	signed char (8-bit integer)
'B'	unsigned char (8-bit integer)
'h'	short (16-bit integer)
'H'	unsigned short (16-bit integer)
'i'	int (32-bit integer)
'I'	unsigned int (32-bit integer)
'l'	long (32 or 64 bit integer)
'L'	unsigned long (32 or 64 bit integer)
'q'	long long (64 bit integer)
'Q'	unsigned long long (64 bit integer)
'f'	float (32 bit)
'd'	double (64 bit)
's'	char[] (String)
'p'	char[] (String with 8-bit length)
'P'	void * (Pointer)

# struct module

- Each code may be preceded by a repeat count

```
'4i' 4 integers
'20s' 20-byte string
```

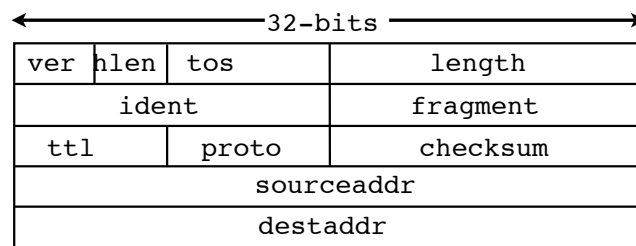
- Integer alignment modifiers

```
'@' Native byte order and alignment
'=' Native byte order, standard alignment
'<' Little-endian, standard alignment
'>' Big-endian, standard alignment
'!' Network (big-endian), standard align
```

- Unless otherwise specified, fields are simply packed together with no alignment or padding

# struct Example

- An IP packet header has this structure



- To unpack in Python, might do this:

```
(vhlen, tos, length,
 ident, fragment,
 ttl, proto, checksum,
 sourceaddr,
 destaddr) = struct.unpack("!BBHHHBBHII", pkt)
```

```
ver = (vhlen & 0xf0) >> 4
hlen = vhlen & 0x0f
```



# Exercise 4.4

(Optional)

Time : 15 Minutes

## collections Module

- An assortment of special-purpose containers
  - deque - Queue
  - Counter - Tabulation/Histograms
  - defaultdict - Dicts with automatic init
  - OrderedDict - Order-preserving dictionary

# deque

- Similar to a list, but append/pop on either end

```
>>> from collections import deque
>>> items = deque([3,4])
>>> items.append(9)
>>> items.appendleft(1)
>>> items
deque([1,3,4,9])
>>>
```

- Removing items

```
>>> items.pop()
9
>>> items.popleft()
1
>>>
```

- Much faster than a normal list for this

# Counters

- Sometimes you want to tabulate (histograms)
- Easy solution: Counter objects

```
from collections import Counter

words = ['yes', 'but', 'no', 'but', 'yes']
wordcounts = Counter(words)
```

- Maps items to an integer count

```
>>> wordcounts['yes']
2
>>> wordcounts['no']
1
>>>
```

# Counters

- Counters can rank things

```
words = ['Look', 'into', 'my', 'eyes', 'look', 'into',
 'my', 'eyes', 'the', 'eyes', 'the', 'eyes',
 'the', 'eyes', 'not', 'around', 'the', 'eyes',
 "don't", 'look', 'around', 'the', 'eyes',
 'look', 'into', 'my', 'eyes', "you're", 'under']
```

```
>>> c = Counter(words)
>>> c.most_common(3)
[('eyes', 8), ('the', 5), ('look', 3)]
>>>
```

- Counters are cool

# defaultdict

- A dict that automatically initializes elements

```
>>> from collections import defaultdict
>>> d = defaultdict(int)
>>> d['x']
0
>>> d['y']
0
defaultdict(<type 'int'>, {'x': 0, 'y': 0})
>>>
```

- Very useful to combine inserts with another operation (e.g., adding)

```
>>> d['z'] += 42
>>> d
defaultdict(<type 'int'>, {'x': 0, 'y': 0, 'z': 42})
>>>
```

# defaultdict

- Example: dict with multiple values per key

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
>>> d['x'].append(1)
>>> d['y'].append(2)
>>> d['x'].append(3)
>>> d
defaultdict(<type 'list'>, {'y': [2], 'x': [1, 3]})
>>> d['x']
[1, 3]
>>>
```

- Automatically creates the initial list into which values are appended

# defaultdict

- Example: Building an index

```
words = ['Look', 'into', 'my', 'eyes', 'look', 'into',
 'my', 'eyes', 'the', 'eyes', 'the', 'eyes',
 'the', 'eyes', 'not', 'around', 'the', 'eyes',
 "don't", 'look', 'around', 'the', 'eyes',
 'look', 'into', 'my', 'eyes', "you're", 'under']
```

```
>>> index = defaultdict(list)
>>> for n, word in enumerate(words):
 index[word].append(n)
```

```
>>> index['eyes']
[3, 7, 9, 11, 13, 17, 22, 26]
>>>
```

# itertools

- Library that provides various iteration patterns

```
>>> import itertools
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]

>>> # Sequence chaining
>>> for x in itertools.chain(a,b):
 print x,

1 2 3 4 5 6
>>>

>>> # Combinations
>>> for x in itertools.combinations(a, 2):
 print x,

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

Copyright (C) 2014, <http://www.dabeaz.com>

4-79

# Data Grouping

- `itertools.groupby(sequence, [key, func])`

```
data = [('1/16/2014', 1, 2),
 ('1/16/2014', 3, 4),
 ('1/16/2014', 5, 6),
 ('1/17/2014', 7, 8),
 ('1/18/2014', 9, 10),
 ('1/18/2014', 11, 12)]

>>> def date(row):
 return row[0]

>>> # Group by date
>>> for date, rows in itertools.groupby(data, get_date):
 print date, ':', len(list(rows)), 'items'

1/16/2014 : 3 rows
1/17/2014 : 1 rows
1/18/2014 : 2 rows
>>>
```

Copyright (C) 2014, <http://www.dabeaz.com>

4-80

# Exercise 4.5

(Optional)

Time : 10 Minutes

## Third Party Modules

- Python has a large library of built-in modules ("batteries included")
- There are even more third party modules
- Python Package Index (PyPi)

<http://pypi.python.org/>

- Or just do a Google search for a topic

# Some Notable Modules

- numpy, scipy : Arrays and vector mathematics
- pandas : Stats and data analysis
- matplotlib : Mathematical plotting
- twisted, gevent : Async I/O and networking
- django, flask : Web programming
- sqlalchemy : Databases and ORM
- ipython : Alternative interactive shell

# Installing Modules

- Installation of a module is likely to take three different forms (depends on the module)
- Platform-native installer
- OS Package Manager (Linux)
- Manual Installation
- setuptools/pip

# Platform Native Install

- You downloaded a third party module as an .exe (PC) or .dmg (Mac) file
- Just run this file and follow installation instructions like you do for installing other software
- You are only likely to see these installers for the more major third-party extensions

# OS Package Manager

- On Linux, you can often install Python extensions using the system package manager
- Example : Ubuntu

```
bash % sudo apt-get install python-packagename
```
- Personal experience: It works pretty well



# Manual Installation

- You downloaded a Python module or package using a standard file format such as a `.gz`, `.tar.gz`, `.tgz`, `.bz2`, or `.zip` file
- Unpack the file and look for a `setup.py` file in the resulting folder
- Run Python on that `setup.py` file

```
bash % python setup.py install
Installation messages
...
```

# pip/setuptools

- There are some third-party package managers
- Most modern and widely used is pip

```
bash % python -m pip install packagename
```

- Older alternative (`easy_install`)

```
bash % easy_install packagename
```

- Pro tip: Use pip

# Commentary

- Installing third party modules is always a delicate matter
- More advanced modules may involve C/C++ code which has to be compiled to native code on your platform.
- May have dependencies on other modules
- In practice, it can be very difficult if you're building the entire environment from source

# Summary

- Have looked at module/package mechanism
- Some of the very basic built-in modules
- How to install third party modules
- We will focus on more of the built-in modules later in the course

# Exercise 4.6

(Optional)

Time : 15 Minutes

Optional Topic

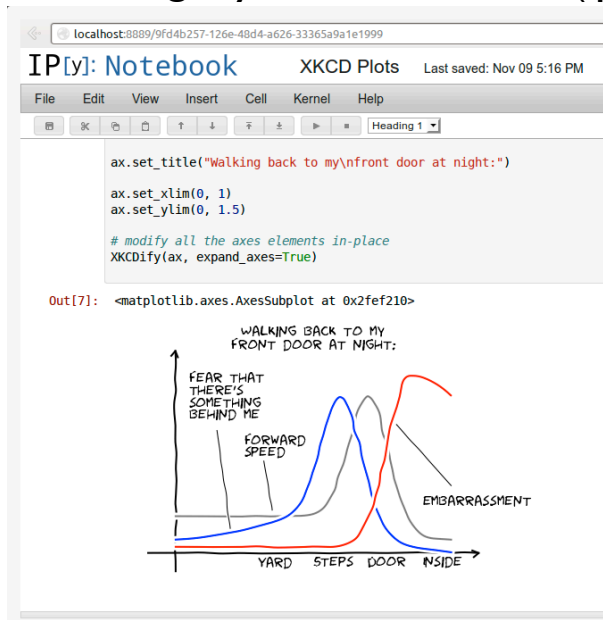
# Introduction to Numpy, Pandas, and Matplotlib

# A Disclaimer

- These are huge modules
- Hundreds (if not thousands) of features
- Reference manual is essential
- Our focus :The "big picture" (e.g., concepts)

# Working Environment

- Consider using IPython Notebook ([ipython.org](http://ipython.org))



# numpy

- A third-party module available here:  
<http://numpy.org/>
- In a nutshell, numpy provides the following
  - A useful N-dimensional array object
  - High-performance operations for manipulating the array data
  - An assortment of numerical algorithms

## Numpy Arrays

- The centerpiece of numpy :The array

```
import numpy
a = numpy.zeros(shape=(M,N), dtype=float)
```

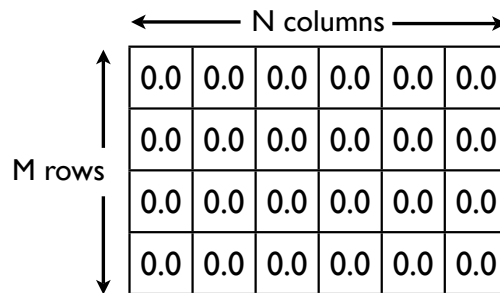
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

- A collection of values like arrays in C/Fortran

# Array Shape

- Arrays have a shape (dimensions)

```
import numpy
a = numpy.zeros(shape=(M,N), dtype=float)
```



- Note : Could have fewer/more dimensions

# Array Shape

- 1-dimension array

```
>>> a = numpy.zeros(shape=5, dtype=float)
>>> a
array([0., 0., 0., 0., 0.])
>>>
```

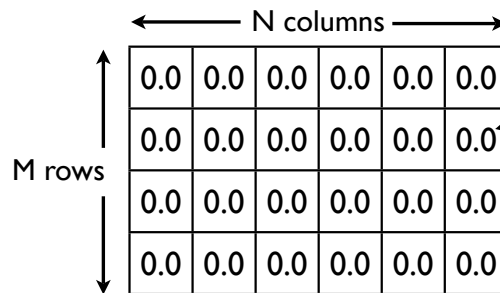
- 3-dimensional array

```
>>> a = numpy.zeros(shape=(5,5,5), dtype=float)
>>> a
array([[[0., 0., 0., 0., 0.],
 ...
 [0., 0., 0., 0., 0.]],
 [[0., 0., 0., 0., 0.],
 ...
 [0., 0., 0., 0., 0.]])
>>>
```

# Array Type

- Arrays have a specified datatype

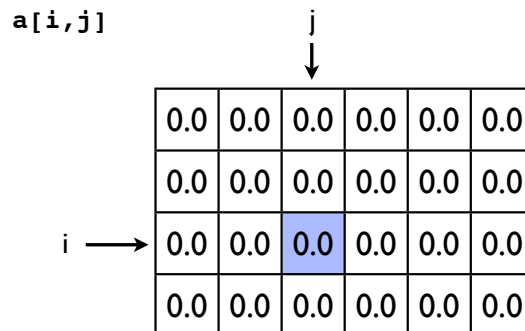
```
import numpy
a = numpy.zeros(shape=(M,N), dtype=float)
```



- Every element must be of that type (no mixed datatypes like a Python list)

# Array Access

- Accessing a single item [row,column]



- Note: different than accessing nested lists

# Array Access

- Accessing an entire row

$a[i]$  →

0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

# Array Access

- Accessing an entire column

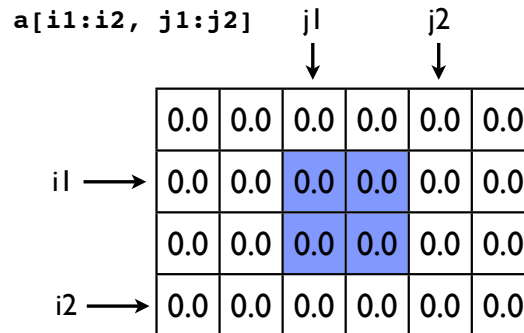
$a[:,j]$   
↓

0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0



# Array Access

- Accessing a region (slice)



- Note: Standard slicing rules apply (the ending index is not included)

# Array Assignment

- Assignments work as long as right hand side can be made to fit

$a[1:3, 2:4] = [[1.0, 2.0], [3.0, 4.0]]$

0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	2.0	0.0	0.0
0.0	0.0	3.0	4.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

- Will get an error if size mismatch

# Array Assignment

- Assignments work as long as right hand side can be made to fit

`a[1:3,2:4] = [1.0 ,2.0]`

Note: value gets broadcast across the rows

0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	2.0	0.0	0.0
0.0	0.0	1.0	2.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

# Array Assignment

- Assignments work as long as right hand side can be made to fit

`a[1:3,2:4] = 1.0`

Assigning a scalar just sets every element to that value

0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	1.0	0.0	0.0
0.0	0.0	1.0	1.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

# Array Math

- Operations with scalars - Apply to all elements

```
>>> a = array([1,2,3,4])
>>> a + 10
array([11, 12, 13, 14])
>>> a * 10
array([10, 20, 30, 40])
>>>
```

- These operations create a new array as a result

# Array Math

- Operations with other arrays

```
>>> a = array([[1,2,3],[4,5,6]])
>>> b = array([[10,11,12],[13,14,15]])
>>> a + b
array([[11, 13, 15],
 [17, 19, 22]])
>>> a * b
array([[10, 22, 36],
 [52, 70, 96]])
>>>
```

- Operations are performed on an element-by-element basis (note: not linear algebra)

# Universal Functions

- Operations that apply to all elements

```
numpy.sin(a)
numpy.cos(a)
numpy.tan(a)
numpy.sqrt(a)
numpy.log(a)
numpy.atan(a)
...
```

- There are more than 60 such functions

- Example:

```
>>> a = numpy.array([1,2,3,4])
>>> b = numpy.sqrt(a)
>>> b
array([1., 1.41421356, 1.73205081, 2.])
>>>
```

# Big Picture

- To best use numpy, you try to apply operations to the entire array (or a large region) at once
- Why? High performance.
- You don't write code like this:

```
for i in xrange(rows):
 for j in xrange(columns):
 a[i,j] +=1 # Increment a single element
```

- Instead, you write this

```
a +=1 # Increment all elements
```

# Array Conditionals

- Conditionals (<,>,<=,>=,==,!=) make arrays

```
>>> a = numpy.array([3,-4,-2,4,5])
>>> b = a < 0
>>> b
array([False, True, True, False, False], dtype=bool)
>>>
```

- `where(condition, x, y)` - Selects values from `x` or `y` depending on the values of `condition`

```
>>> numpy.where(b, a, 0)
array([0, -4, -2, 0, 0])
>>>
```

# Array Conditionals

- Example : Evaluate this function on an array

```
def f(x):
 if x < 0:
 return (1-x)
 else:
 return cos(x)
```

- Solution:

```
>>> xvalues = numpy.arange(-5,6)
>>> xvalues
array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])
>>> yvalues = numpy.where(xvalues < 0,
 1-xvalues,
 numpy.cos(xvalues))
>>> yvalues
array([6., 5., 4., 3., 2., 1., 0.54030231, -0.41614684,
 -0.9899925 , -0.65364362, 0.28366219])
>>>
```

# Matrices

- numpy provides a matrix object to support linear algebra operations

```
>>> from numpy import matrix
>>> a = matrix([[1,2,3],[4,5,6],[7,8,9]])
```

- Operations with scalars work as before

```
>>> a * 3
matrix([[3, 6, 9],
 [12, 15, 18],
 [21, 24, 27]])
>>> a + 10
matrix([[11, 12, 13],
 [14, 15, 16],
 [17, 18, 19]])
>>>
```

# Matrices

- Operations involving other matrices follow standard library algebra rules

```
>>> a = matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> b = matrix([[10,11,12]])
>>> a * b
Traceback (most recent call last):
...
ValueError: objects are not aligned
>>> b * a
matrix([[138, 171, 204]])
>>>
```

- There are many other operations (eigenvalues, svd decomposition, qr decomposition, etc.)
- However, this isn't a linear algebra masterclass

# Exercise data. I

Time : 15 Minutes

# Pandas

- Python Data Analysis Library  
<http://pandas.pydata.org>
- Provides
  - Data analysis functionality
  - Stats functions
  - Various file formats (CSV, Excel, etc.)

# Dataframes

- Centerpiece of Pandas is a "Dataframe"
- A collection of typed columns

## CSV

```
name,shares,price
"AA",100,32.20
"IBM",50,91.10
"CAT",150,83.44
"MSFT",200,51.23
"GE",95,40.37
"MSFT",50,65.10
"IBM",100,70.44
```

`pandas.read_csv()`



## dataframe

```
name shares price
(object) (int64) (float64)
0 AA 100 32.20
1 IBM 50 91.10
2 CAT 150 83.44
3 MSFT 200 51.23
4 GE 95 40.37
5 MSFT 50 65.10
6 IBM 100 70.44
```

- Each column is a "series" (like a numpy array)

# Dataframes

- Think spreadsheets...

```
>>> df = pandas.read_csv('portfolio.csv')
>>> df
 name shares price
0 AA 100 32.20
1 IBM 50 91.10
2 CAT 150 83.44
3 MSFT 200 51.23
4 GE 95 40.37
5 MSFT 50 65.10
6 IBM 100 70.44
>>>
```

- Columns and rows



# Dataframes

- Column access (use the column header)

```
>>> df['shares']
0 100
1 50
2 150
3 200
4 95
5 50
6 100
Name: shares, dtype: int64
>>>
```

- Accessing a specific row item

```
>>> df['shares'][4]
95
>>>
```

# Filtering

- Filtering on a column

```
>>> df['name'] == 'IBM'
0 False
1 True
2 False
3 False
4 False
5 False
6 True
Name: name, dtype: bool
```

```
>>> df[df['name'] == 'IBM']
 name shares price
1 IBM 50 91.10
6 IBM 100 70.44
>>>
```

```
>>> df['shares'] > 100
0 False
1 False
2 True
3 True
4 False
5 False
6 False
Name: shares, dtype: bool
```

```
>>> df[df['shares'] > 100]
 name shares price
2 CAT 150 83.44
3 MSFT 200 51.23
>>>
```

# Sorting

```
>>> df.sort('name')
 name shares price
0 AA 100 32.20
2 CAT 150 83.44
4 GE 95 40.37
1 IBM 50 91.10
6 IBM 100 70.44
3 MSFT 200 51.23
5 MSFT 50 65.10
```

```
>>> df.sort('shares')
 name shares price
1 IBM 50 91.10
5 MSFT 50 65.10
4 GE 95 40.37
0 AA 100 32.20
6 IBM 100 70.44
2 CAT 150 83.44
3 MSFT 200 51.23
>>>
```

```
>>> df.sort('shares',
 ascending=False)
 name shares price
3 MSFT 200 51.23
2 CAT 150 83.44
6 IBM 100 70.44
0 AA 100 32.20
4 GE 95 40.37
5 MSFT 50 65.10
1 IBM 50 91.10
>>>
```

# Grouping

```
>>> for name, frame in df.groupby('name'):
... print ':::', name
... print frame
...
::: AA
 name shares price
0 AA 100 32.2
::: CAT
 name shares price
2 CAT 150 83.44
::: GE
 name shares price
4 GE 95 40.37
::: IBM
 name shares price
1 IBM 50 91.10
6 IBM 100 70.44
::: MSFT
 name shares price
3 MSFT 200 51.23
5 MSFT 50 65.10
```

# Grouping/Aggregation

```
>>> total_shares = df.groupby('name')['shares'].sum()
>>> total_shares
name
AA 100
CAT 150
GE 95
IBM 150
MSFT 250
Name: shares, dtype: int64

>>> total_shares['MSFT']
250
>>>
```

# Comments

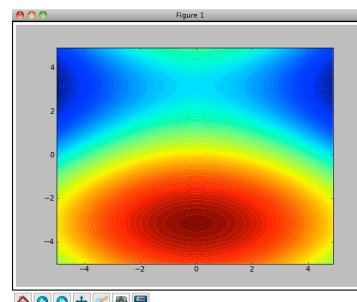
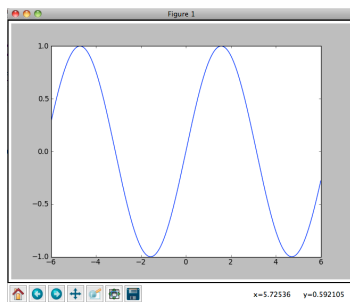
- Pandas is a little weird at first
- Very powerful
- An alternative to list comprehensions and similar list processing techniques
- Inspired more by R than typical Python idioms

# Exercise data.2

Time : 15 Minutes

# matplotlib

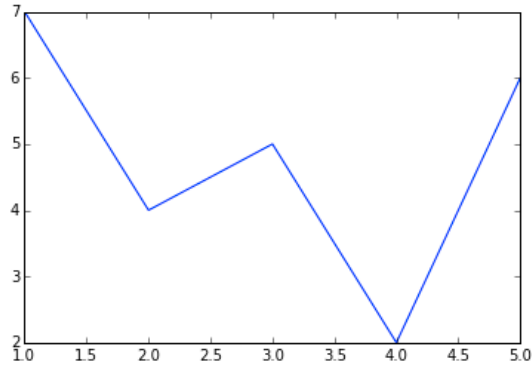
- A popular third-party module for making plots  
<http://matplotlib.sourceforge.net>



# Matplotlib

- Example of a simple x-y plot

```
>>> import matplotlib.pyplot as plt
>>> xpts = [1, 2, 3, 4, 5]
>>> ypts = [7, 4, 5, 2, 6]
>>> plot(xpts, ypts)
[<matplotlib.lines.Line2D object at 0x2fc2f10>]
>>>
```



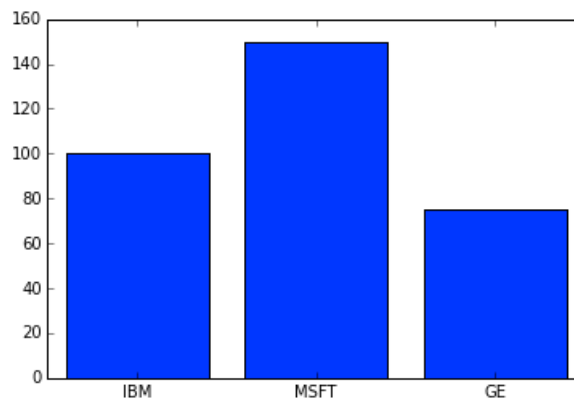
Copyright (C) 2014, <http://www.dabeaz.com>

4-127

# Matplotlib

- Example of a bar plot

```
>>> import matplotlib.pyplot as plt
>>> shares = { 'IBM': 100, 'MSFT':150, 'GE':75 }
>>> xpts = range(len(shares))
>>> plt.bar(xpts, shares.values(), align='center')
>>> plt.xticks(xpts, list(shares))
```



Copyright (C) 2014, <http://www.dabeaz.com>

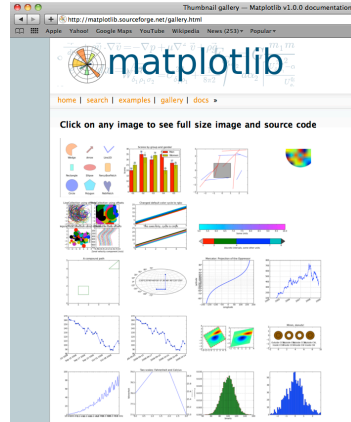
4-128

# Using Matplotlib

- Go to the "gallery" page

<http://matplotlib.sourceforge.net/gallery.html>

- Click on a plot that is similar to what you want to make
- See a code sample
- Copy/adapt it



## Exercise data.3

Time : 10 Minutes

## Section 5

# Classes and Objects

## OO in a Nutshell

- A programming technique where code is organized as a collection of "objects"
- An "object" consists of
  - Data (attributes)
  - Methods (functions applied to object)
- You've already been doing it

# OO in a Nutshell

- Example: Lists

```
>>> nums = [1, 2, 3]
>>> nums.append(4) # Method
>>> nums.insert(1,10) # Method
>>>
```

- `nums` is an "instance" of a list
- methods are attached to the instance

# The class statement

- How to define your own custom objects

```
class Circle(object):
 def __init__(self, radius):
 self.radius = radius

 def area(self):
 return math.pi * (self.radius ** 2)

 def perimeter(self):
 return 2 * math.pi * self.radius
```

- What is a class?
- It's a collection of functions that perform various operations on instances



# Instances

- Created by calling the class as a function

```
>>> c = Circle(4.0)
>>> d = Circle(5.0)
>>>
```

- Each instance has its own data

```
>>> c.radius
4.0
>>> d.radius
5.0
>>>
```

- You invoke methods on instances to do things

```
>>> c.area()
50.26548245743669
>>> d.perimeter()
31.415926535897931
>>>
```

## \_\_init\_\_ method

- This method initializes a new instance
- Called whenever a new object is created

```
>>> c = Circle(4.0)

class Circle(object):
 def __init__(self, radius):
 self.radius = radius
```

newly created object

- `__init__` is example of a "special method"
- Has special meaning to Python interpreter

# Instance Data

- Each instance has its own data (attributes)

```
class Circle(object):
 def __init__(self, radius):
 self.radius = radius
```

- Inside methods, you refer to this data using self

```
def area(self):
 return math.pi * (self.radius ** 2)
```

- In other code, you just use the variable that you're using to name the instance

```
>>> c = Circle(4.0)
>>> c.radius
4.0
```

# Methods

- Functions applied to instances of an object

```
class Circle(object):
 ...
 def area(self):
 return math.pi * (self.radius ** 2)
```

- The object is always passed as first argument

```
>>> c.area()
 ↙
def area(self):
 ...
```

- By convention, the instance is called "self"

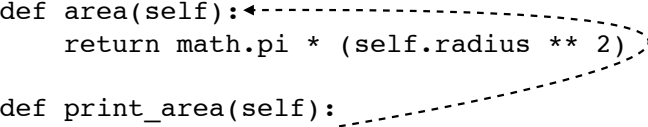
The name is unimportant---the object is always passed as the first argument. It is simply Python programming style to call this argument "self." It's similar to "this" in C++/Java.

# Calling Other Methods

- Methods call other methods via self

```
class Circle(object):
 def area(self):
 return math.pi * (self.radius ** 2)

 def print_area(self):
 print self.area()
```



- A caution : Code like this doesn't work

```
class Circle(object):
 ...
 def print_area(self):
 print area() # ! Error
```

- This merely calls a global function area()

## Exercise 5.1

Time : 15 Minutes

# Inheritance

- A tool for specializing objects

```
class Parent(object):
 ...

class Child(Parent):
 ...
```

- New class called a derived class or subclass
- Parent known as base class or superclass
- Parent is specified in () after class name

# Inheritance

- What do you mean by "specialize?"
- Take an existing class and ...
  - Add new methods
  - Redefine some of the existing methods
  - Add new attributes to instances

# Inheritance Example

- In bill #246 of the 1897 Indiana General Assembly, there was text that dictated a new method for squaring a circle, which if adopted, would have equated  $\pi$  to 3.2.
- Fortunately, it was never adopted because an observant mathematician took notice...
- But, let's make a special Indiana Circle anyways...

# Inheritance Example

- Specializing a class

```
class INCircle(Circle):
 def area(self):
 return 3.2 * (self.radius ** 2)
```

- Using the specialized version

```
>>> c = INCircle(4.0) # Calls Circle.__init__
>>> c.radius
4.0
>>> c.area() # Calls INCircle.area
51.20
>>> c.perimeter() # Calls Circle.perimeter
25.132741228718345
>>>
```

- It's the same as Circle except for area()

# Using Inheritance

- Inheritance sometimes used to organize objects

```
class Shape(object):
 ...

class Circle(Shape):
 ...

class Rectangle(Shape):
 ...
```

- Think of a logical hierarchy or taxonomy

# Using Inheritance

- More commonly used as a code reuse tool

```
class CustomHandler(TCPHandler):
 def handle_request(self):
 ...
 # Custom processing
```

- Base class contains general purpose code
- You inherit to customize specific parts
- Maybe it plugs into a framework

# "is a" relationship

- Inheritance establishes a type relationship

```
class Shape(object):
 ...

class Circle(Shape):
 ...

>>> c = Circle(4.0)
>>> isinstance(c, Shape)
True
>>>
```

- Important: objects defined via inheritance are supposed to be interchangeable with the parent

# object base class

- If a class has no parent, use object as base

```
class Foo(object):
 ...
```

- object is the parent of all objects in Python
- Note : Sometimes you will see code where classes are defined without any base class. That is an older style of Python coding that has been deprecated for almost 15 years. When defining a new class, you always inherit from something.

# Inheritance and Overriding

- Sometimes a class extends an existing method, but it has to use the original implementation

```
class Foo(object):
 def spam(self):
 ...
class Bar(Foo):
 def spam(self):
 ...
 r = super(Bar, self).spam()
 ...
```

notice how both methods have the same name.

- Use `super()` to do it
- Admittedly, it looks a little funny

# Inheritance and `__init__`

- With inheritance, you must initialize parents

```
class Shape(object):
 def __init__(self):
 self.x = 0.0
 self.y = 0.0
 ...
class Circle(Shape):
 def __init__(self, radius):
 super(Circle, self).__init__() # init base
 self.radius = radius
```

- Again, you should use `super()` as shown



# Overriding Caution

- Sometimes you will see overrides implemented with a direct call to the parent class

```
class Foo(object):
 def spam(self):
 ...
 ...
class Bar(Foo):
 def spam(self):
 ...
 r = Foo.spam(self)
 ...
```

direct call to parent

- This is an older style with subtle limitations
- Usually better to use `super()` instead

# Calling Other Methods

- With inheritance, the correct method gets called if overridden (depends on the type of `self`)

```
class Circle(object):
 def area(self):
 return math.pi * (self.radius ** 2)

 def print_area(self):
 print self.area()

class INCircle(Circle):
 def area(self):
 return 3.2 * (self.radius ** 2)
```

if `self` is an instance of `INCircle`

- Example:

```
>>> c = INCircle(4)
>>> c.print_area()
51.2
>>>
```

# Multiple Inheritance

- You can specifying multiple base classes

```
class Foo(object):
 ...
class Bar(object):
 ...
class Spam(Foo, Bar):
 ...
```

- The new class inherits features from both parents
- But there are some tricky details (later)
- Don't do it unless you know what you're doing

## Exercise 5.2

Time : 30 Minutes

# Special Methods

- Classes may define special methods
- Have special meaning to Python interpreter
- Always preceded/followed by `__`

```
class Foo(object):
 def __init__(self):
 ...

 def __del__(self):
 ...
```

- There are several dozen special methods
- Will show a few examples

# String Conversions

- Objects have two string representations

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> print d
2012-12-21
>>> d
datetime.date(2012, 12, 21)
>>>
```

- `str(x)` - Printable output

```
>>> str(d)
'2012-12-21'
>>>
```

- `repr(x)` - For programmers

```
>>> repr(d)
'datetime.date(2012, 12, 21)'
>>>
```

# String Conversions

```
class Date(object):
 def __init__(self, year, month, day):
 self.year = year
 self.month = month
 self.day = day

 def __str__(self):
 return '%d-%d-%d' % (self.year,
 self.month,
 self.day)

 def __repr__(self):
 return 'Date(%r,%r,%r)' % (self.year,
 self.month,
 self.day)
```

Note: The convention for `__repr__()` is to return a string that, when fed to `eval()`, will recreate the underlying object. If this is not possible, some kind of easily readable representation is used instead.

Copyright (C) 2014, <http://www.dabeaz.com>

5-27

# Methods: Mathematics

- Mathematical operators

<code>a + b</code>	<code>a.__add__(b)</code>
<code>a - b</code>	<code>a.__sub__(b)</code>
<code>a * b</code>	<code>a.__mul__(b)</code>
<code>a / b</code>	<code>a.__div__(b)</code>
<code>a // b</code>	<code>a.__floordiv__(b)</code>
<code>a % b</code>	<code>a.__mod__(b)</code>
<code>a &lt;&lt; b</code>	<code>a.__lshift__(b)</code>
<code>a &gt;&gt; b</code>	<code>a.__rshift__(b)</code>
<code>a &amp; b</code>	<code>a.__and__(b)</code>
<code>a   b</code>	<code>a.__or__(b)</code>
<code>a ^ b</code>	<code>a.__xor__(b)</code>
<code>a ** b</code>	<code>a.__pow__(b)</code>
<code>-a</code>	<code>a.__neg__()</code>
<code>~a</code>	<code>a.__invert__()</code>
<code>abs(a)</code>	<code>a.__abs__()</code>

- Consult reference for further details

Copyright (C) 2014, <http://www.dabeaz.com>

5-28

# Methods: Item Access

- Methods used to implement containers

```
len(x) x.__len__()
x[a] x.__getitem__(a)
x[a] = v x.__setitem__(a,v)
del x[a] x.__delitem__(a)
```

- Use in a class

```
class Sequence(object):
 def __len__(self):
 ...
 def __getitem__(self,a):
 ...
 def __setitem__(self,a,v):
 ...
 def __delitem__(self,a):
 ...
```

# Odds and Ends

- Defining new exceptions
- Bound and unbound methods
- Alternative attribute lookup

# Defining Exceptions

- User-defined exceptions are defined by classes

```
class NetworkError(Exception):
 pass
```

- Exceptions always inherit from Exception
- Usually, it's just an empty class (use pass)
- You can also make a hierarchy

```
class AuthenticationError(NetworkError):
 pass

class ProtocolError(NetworkError):
 pass
```

# Method Invocation

- Invoking a method is a two-step process
- Lookup: The . operator
- Method call: The () operator

```
class Stock(object):
 ...
 def cost(self):
 return self.shares*self.price

>>> s = Stock('GOOG', 100, 490.10)
>>> c = s.cost ← Lookup
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()
49010.0 ← Method call
>>>
```

# Bound Methods

- A method that has not yet been invoked by the function call operator () is known as a "bound method"
- It operates on the instance where it originated

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s
<Stock object at 0x590d0>
>>> c = s.cost
>>> c
<bound method Stock.cost of <Stock object at 0x590d0>>
>>> c()
49010.0
>>>
```

← binding →

# Bound Methods

- Why would you care?
- Often a source of careless non-obvious errors

```
>>> s = Stock('GOOG', 100, 490.10)
>>> print "Cost : %0.2f" % s.cost
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: float argument required, not instancemethod
>>>
```

← Note missing ()

- Or devious behavior that's hard to debug

```
f = open(filename, 'w')
...
f.close
```

← Oops. Didn't do anything at all

# Attribute Access

- These functions may be used to manipulate attributes given an attribute name string

```
getattr(obj, 'name') # Same as obj.name
setattr(obj, 'name', value) # Same as obj.name = value
delattr(obj, 'name') # Same as del obj.name
hasattr(obj, 'name') # Tests if attribute exists
```

- Example: Probing for an optional attribute

```
if hasattr(obj, 'x'):
 x = getattr(obj, 'x'):
else:
 x = None
```

- Note: `getattr()` has a useful default value arg

```
x = getattr(obj, 'x', None)
```

# Summary

- A high-level overview of classes
- Most code involving classes will involve the topics covered in this section
- If you're merely using existing libraries, the code is typically fairly simple



# Exercise 5.3

Time : 15 Minutes

## Section 6

# The Inner Workings of Python Objects

## Overview

- A few more details about how objects work
- How objects are represented
- Details of attribute access
- Data encapsulation

# Dictionaries Revisited

- A dictionary is a collection of named values

```
stock = {
 'name' : 'GOOG',
 'shares' : 100,
 'price' : 490.10
}
```

- Dictionaries are commonly used for simple data structures (shown above)
- However, they are used for critical parts of the interpreter and may be the most important type of data in Python

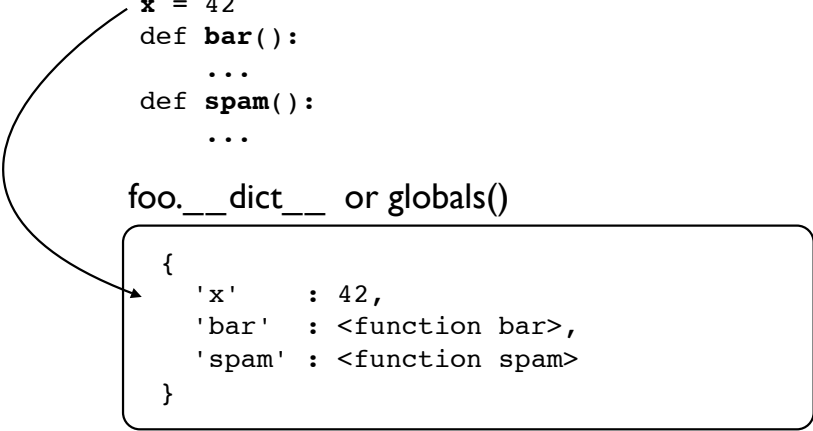
# Dicts and Modules

- In a module, a dictionary holds all of the global variables and functions

```
foo.py
```

```
x = 42
def bar():
 ...
def spam():
 ...
```

foo.\_\_dict\_\_ or globals()



```
{
 'x' : 42,
 'bar' : <function bar>,
 'spam' : <function spam>
}
```

# Dicts and Objects

- User-defined objects also use dictionaries
  - Instance data
  - Class members
- In fact, the entire object system is mostly just an extra layer that's put on top of dictionaries
- Let's take a look...

# Dicts and Instances

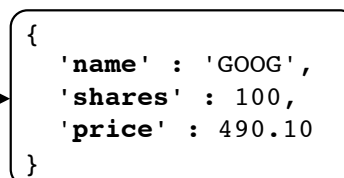
- A dictionary holds instance data (`__dict__`)

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s.__dict__
{'name' : 'GOOG', 'shares' : 100, 'price': 490.10 }
```

- You populate this dict when assigning to self

```
class Stock(object):
 def __init__(self, name, shares, price):
 self.name = name
 self.shares = shares
 self.price = price
```

`self.__dict__` →



instance data

# Dicts and Instances

- Critical point : Each instance gets its own private dictionary

```
s = Stock('GOOG',100,490.10)
t = Stock('AAPL',50,123.45)
```

```
{
 'name' : 'GOOG',
 'shares' : 100,
 'price' : 490.10
}
```

- So, if you created 100 instances of some class, there are 100 dictionaries sitting around holding data

```
{
 'name' : 'AAPL',
 'shares' : 50,
 'price' : 123.45
}
```

# Dicts and Classes

- A dictionary holds the methods of a class

```
class Stock(object):
 def __init__(self,name,shares,price):
 self.name = name
 self.shares = shares
 self.price = price
 def cost(self):
 return self.shares*self.price
 def sell(self,nshares):
 self.shares -= nshares
```

Stock.\_\_dict\_\_

```
{
 'cost' : <function>,
 'sell' : <function>,
 '__init__' : <function>,
}
```

methods

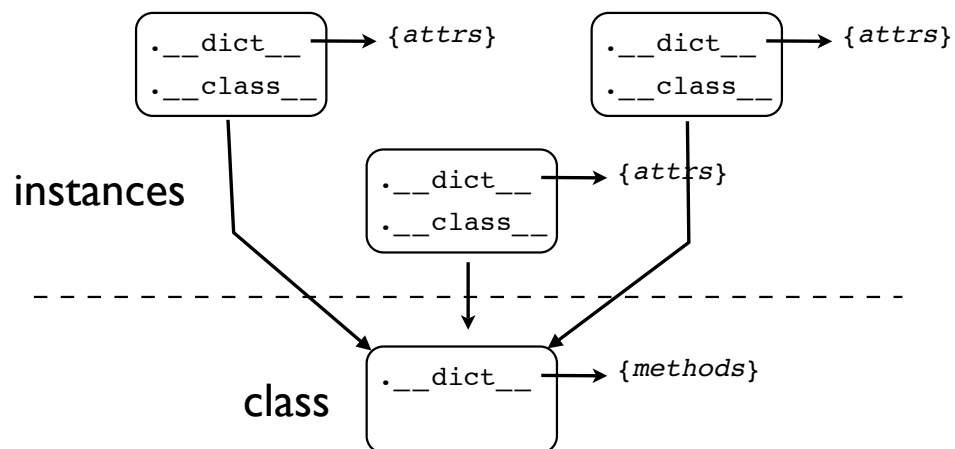
# Instances and Classes

- Instances and classes are linked together
- `__class__` attribute refers back to the class

```
>>> s = Stock('GOOG', 100, 490.10)
>>> s.__dict__
{'name': 'GOOG', 'shares': 100, 'price': 490.10 }
>>> s.__class__
<class '__main__.Stock'>
>>>
```

- The instance dictionary holds data unique to each instance whereas the class dictionary holds data collectively shared by all instances

# Instances and Classes



# Attribute Access

- When you work with objects, you access data and methods using the (.) operator

```
x = obj.name # Getting
obj.name = value # Setting
del obj.name # Deleting
```

- These operations are directly tied to the dictionaries sitting underneath the covers

# Modifying Instances

- Operations that modify an object always update the underlying dictionary

```
>>> s = Stock('GOOG',100,490.10)
>>> s.__dict__
{'name':'GOOG', 'shares':100, 'price':490.10 }
→ >>> s.shares = 50
→ >>> s.date = "6/7/2007"
>>> s.__dict__
{'name':'GOOG', 'shares':50, 'price':490.10,
 'date':'6/7/2007'}
→ >>> del s.shares
>>> s.__dict__
{'name':'GOOG', 'price':490.10, 'date':'6/7/2007'}
>>>
```

# Modifying Instances

- It may be surprising that instances can be extended after creation
- You can freely change attributes at any time

```
>>> s = Stock('GOOG',100,490.10)
>>> s.blah = "some new attribute"
>>> del s.name
>>>
```

- Again, you're just manipulating a dictionary
- Very different from C++/Java where the structure of an object is rigidly fixed

# Reading Attributes

- Suppose you read an attribute on an instance

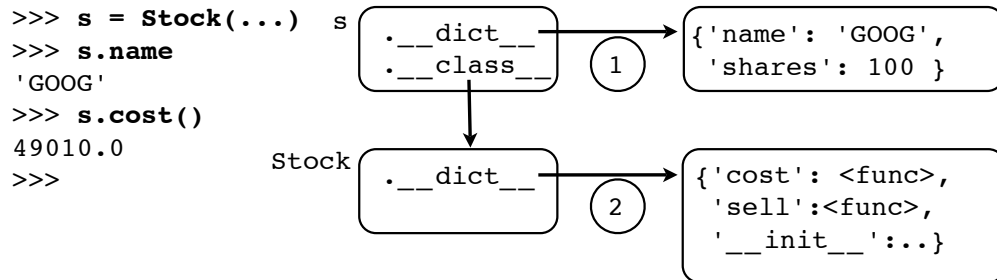
```
x = obj.name
```

- Attribute may exist in two places
  - Local instance dictionary
  - Class dictionary
- So, both dictionaries may be checked



# Reading Attributes

- First check in local `__dict__`
- If not found, look in `__dict__` of class



- This lookup scheme is how the members of a class get shared by all instances

## Exercise 6.1

Time : 10 Minutes

# How Inheritance Works

- Classes may inherit from other classes

```
class A(B,C):
 ...
```

- Bases are stored as a tuple in each class

```
>>> A.__bases__
(<class '__main__.B'>,<class '__main__.C'>)
>>>
```

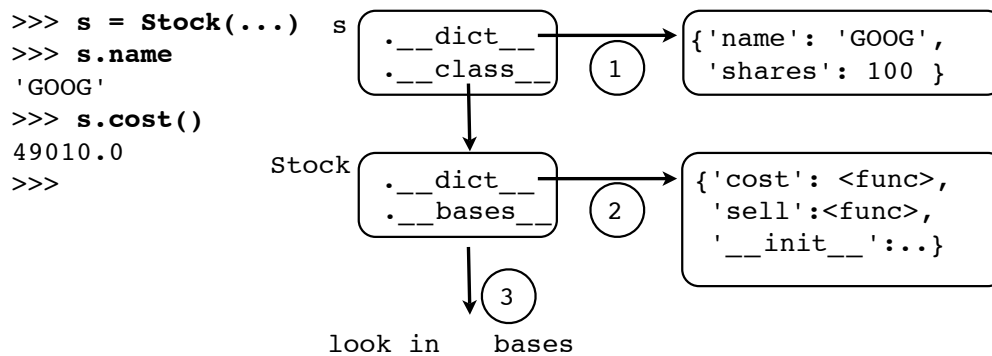
- This provides a link to parent classes
- This link simply extends the search process used to find attributes

Copyright (C) 2014, <http://www.dabeaz.com>

6-17

# Reading Attributes

- First check in local `__dict__`
- If not found, look in `__dict__` of class
- If not found in class, look in base classes



Copyright (C) 2014, <http://www.dabeaz.com>

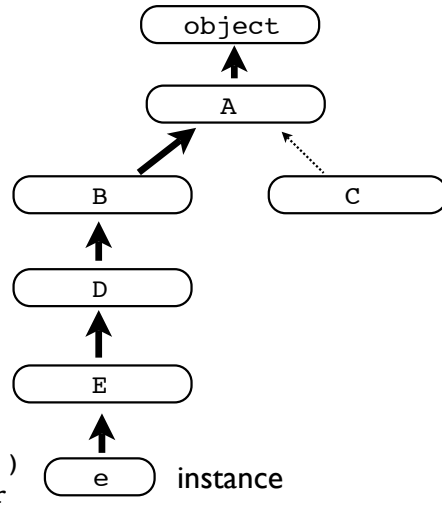
6-18

# Single Inheritance

- In inheritance hierarchies, attributes are found by walking up the inheritance tree

```
class A(object): pass
class B(A): pass
class C(A): pass
class D(B): pass
class E(D): pass
```

- With single inheritance, there is a single path to the top
- You stop with the first match



Copyright (C) 2014, <http://www.dabeaz.com>

6-19

# The MRO

- The inheritance chain is precomputed and stored in an "MRO" attribute on the class

```
>>> E.__mro__
(<class '__main__.E'>, <class '__main__.D'>,
 <class '__main__.B'>, <class '__main__.A'>,
 <type 'object'>)
>>>
```

- "Method Resolution Order"
- To find attributes, Python walks the MRO
- First match wins

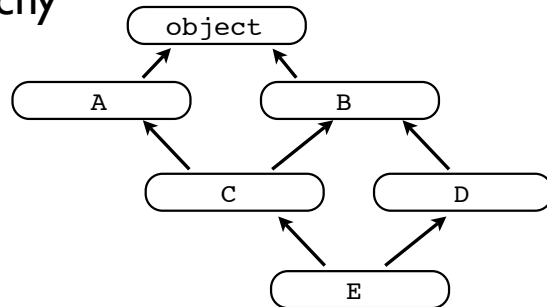
Copyright (C) 2014, <http://www.dabeaz.com>

6-20

# Multiple Inheritance

- Consider this hierarchy

```
class A(object): pass
class B(object): pass
class C(A,B): pass
class D(B): pass
class E(C,D): pass
```



- What happens here?

```
e = E()
e.attr
```

- A similar search process is carried out, but what is the order? That is a problem.

# Multiple Inheritance

- Python uses "cooperative multiple inheritance"
- Big picture: Child classes can arrange their parents to cooperate with each other
- But there are some rules...

Rule 1: Children before parents  
Rule 2: Parents go in order

# Multiple Inheritance

Rule 1: Children before parents

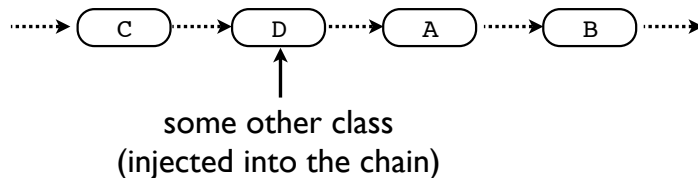
Rule 2: Parents go in order

```
class C(A, B):
 ...
```

search order



- Head explosion: Python might check other classes in-between. This is allowed by the rules.



Copyright (C) 2014, <http://www.dabeaz.com>

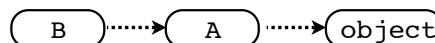
6-23

# Multiple Inheritance

```
class A(object):
 def yow(self):
 print 'Yow!'
```

```
class B(A):
 def spam(self):
 self.yow()
```

search order



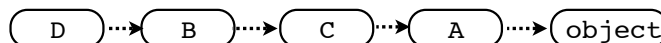
```
>>> b = B()
>>> b.spam()
Yow!
>>>
```

- Now consider

```
class C(A):
 def yow(self):
 print 'Yowzer!!'
```

```
class D(B,C):
 pass
```

search order



```
>>> d = D()
>>> d.spam()
Yowzer!!
>>>
```

- Why? The rules

Copyright (C) 2014, <http://www.dabeaz.com>

6-24

# Multiple Inheritance

- Python flattens the inheritance hierarchy

```
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>,
 <class '__main__.C'>, <class '__main__.A'>,
 <type 'object'>)
>>>
```

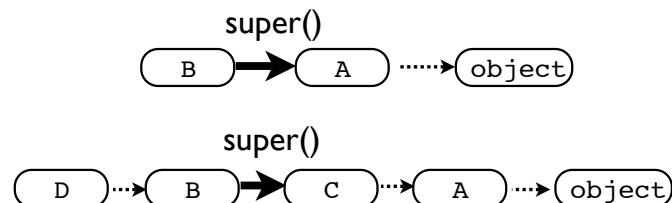
- Calculated using the C3 Linearization algorithm
- Merge of parent MROs according to the "rules"
- Attributes found by walking the MRO as before

# Why super()?

- Always use super() when overriding methods

```
class B(A):
 def foo(self):
 ...
 return super(B,self).foo()
```

- super() delegates to the next class on the MRO



- Tricky bit: You don't know what it is

# Some Cautions

- Multiple inheritance is a powerful tool
- With power comes responsibility
- Frameworks/libraries sometimes use it for advanced features involving composition of components
- More details in an advanced course

# Classes and Encapsulation

- One of the primary roles of a class is to encapsulate data and internal implementation details of an object
- However, a class also defines a "public" interface that the outside world is supposed to use to manipulate the object
- This distinction between implementation details and the public interface is important

# A Problem

- In Python, almost everything about classes and objects is "open"
  - You can easily inspect object internals
  - You can change things at will
  - There's no strong notion of access-control (i.e., private class members)
- If you're trying to cleanly separate the internal "implementation" from the "interface" this becomes an issue

# Python Encapsulation

- Python relies on programming conventions to indicate the intended use of something
- Typically, this is based on naming
- There is a general attitude that it is up to the programmer to observe the rules as opposed to having the language enforce rules



# Private Attributes

- Any attribute name with a leading `_` is considered to be "private"

```
class Person(object):
 def __init__(self, name):
 self._name = 0
```

- However, this is only a programming style
- You can still access it

```
>>> p = Person('Guido')
>>> p._name
'Guido'
>>> p._name = 'Dave'
>>>
```

# Private Attributes

- Variant :Attribute names with two leading `_`

```
class Person(object):
 def __init__(self, name):
 self.__name = name
```

- This kind of attribute is "more private"

```
>>> p = Person('Guido')
>>> p.__name
AttributeError: 'Person' object has no attribute '__name'
>>>
```

- This is actually just a name mangling trick

```
>>> p = Person('Guido')
>>> p._Person__name
'Guido'
>>>
```

# Private Attributes

- Discussion: What style to use?
- Most experienced Python programmers seem to use a single underscore
- Many consider the use of double underscores to cause more problems than they solve
- Example: `getattr()`, `setattr()` don't work right
- You mileage might vary...

# Problem: Simple Attributes

- Consider the following class

```
class Stock(object):
 def __init__(self, name, shares, price):
 self.name = name
 self.shares = shares
 self.price = price
```

```
s = Stock('GOOG', 100, 490.1)
s.shares = 50
```

- Suppose you later wanted to add validation

```
s.shares = "50" # --> TypeError
```

- How would you do it?

# Managed Attributes

- You might introduce accessor methods

```
class Stock(object):
 def __init__(self, name, shares, price):
 self.name = name
 self.set_shares(shares)
 self.price = price

 def get_shares(self):
 return self._shares

 def set_shares(self, value):
 if not isinstance(value, int):
 raise TypeError('Expected an int')
 self._shares = value
```

functions that layer get/  
set operations on top of  
a private attribute

- Too bad this breaks all existing code

`s.shares = 50`       $\longrightarrow$       `s.set_shares(50)`

# Properties

- An alternative approach to accessor methods

```
class Stock(object):
 def __init__(self, name, shares, price):
 self.name = name
 self.shares = shares
 self.price = price

 @property
 def shares(self):
 return self._shares

 @shares.setter
 def shares(self, value):
 if not isinstance(value, int):
 raise TypeError('Expected int')
 self._shares = value
```

- The syntax is a little jarring at first

# Properties

- Normal attribute access triggers the methods

```
class Stock(object):
 def __init__(self, name, shares, price):
 self.name = name
 self.shares = shares
 self.price = price

 @property
 def shares(self):
 return self._shares

 @shares.setter
 def shares(self, value):
 if not isinstance(value, int):
 raise TypeError('Expected int')
 self._shares = value
```

get

set

```
>>> s = Stock(...)
>>> s.shares
100
>>> s.shares = 50
>>>
```

- No changes needed to other source code

# Properties

- You don't change existing attribute access

```
class Stock(object):
 def __init__(self, name, shares, price):
 ...
 self.shares = shares
 ...
 @property
 def shares(self):
 return self._shares

 @shares.setter
 def shares(self, value):
 if not isinstance(value, int):
 raise TypeError('Expected int')
 self._shares = value
```

assignment calls the setter

- Common confusion: property vs private name

# Properties

- Properties are also useful if you are creating objects where you want to have a very consistent user interface
- Example : Computed data attributes

```
class Circle(object):
 def __init__(self, radius):
 self.radius = radius
 @property
 def area(self):
 return math.pi * (self.radius ** 2)
 @property
 def perimeter(self):
 return 2 * math.pi * self.radius
```

# Properties

- Example use:

```
>>> c = Circle(4)
>>> c.radius ← Instance Variable
4
>>> c.area ← Computed Properties
50.26548245743669
>>> c.perimeter ← Computed Properties
25.132741228718345
```

- Commentary : Notice how there is no obvious difference between the attributes as seen by the user of the object

# Uniform Access

- The last example shows how to put a more uniform interface on an object. If you don't do this, an object might be confusing to use:

```
>>> c = Circle(4.0)
>>> a = c.area() # Method
>>> r = c.radius # Data attribute
>>>
```

- Why is the () required for the area, but not for the radius?

# Decorator Syntax

- The @ syntax is known as "decoration"
- Specifies a modifier that's applied to a function definition that immediately follows

```
class Circle(object):
 ...
 @property
 def area(self):
 return math.pi*self.radius**2
 ...
```

- It's kind of like a macro. More details are found in Section 10 (Advanced Topics)

# Properties and Accessors

- Sometimes you may want both accessor functions and properties at the same time

```
class Stock(object):
 def __init__(self, name, shares, price):
 ...
 self.shares = shares
 ...
 def get_shares(self):
 return self._shares

 def set_shares(self, value):
 if not isinstance(value, int):
 raise TypeError('Expected int')
 self._shares = value

 shares = property(get_shares, set_shares)
```

- Use the `property()` function as shown to do it

Copyright (C) 2014, <http://www.dabeaz.com>

6-43

## `__slots__` Attribute

- You can restrict the set of attribute names

```
class Stock(object):
 __slots__ = ('name', '_shares', 'price')
 ...
```

- Produces errors for other attributes

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.price = 385.15
>>> s.prices = 410.2
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
AttributeError: 'Stock' object has no attribute 'prices'
```

- Prevents errors, restricts usage of objects (but is actually used for performance)

Copyright (C) 2014, <http://www.dabeaz.com>

6-44

# Commentary

- Don't go overboard with private attributes, properties, slots, etc.
- They serve a specific purpose and you may see them when reading other's Python code
- Not necessary for most day-to-day coding

## Exercise 6.2

Time : 10 Minutes



## Section 7

# Testing and Debugging

## Overview

- Testing
- Logging, error handling, and diagnostics
- Debugging
- Profiling

# Testing Rocks, Debugging Sucks

- What else is there to say?
- Dynamic nature of Python makes testing critically important to most applications
- There is no compiler to find your bugs
- Only way to find bugs is to run the code and make sure you exercise all of its features

# Testing Modules

- There are two core modules for testing
  - doctest
  - unittest
- Programmers tend to use both, but for different purposes

# Testing: doctest module

- A module that runs interactive examples embedded inside documentation strings

```
simple.py
def add(x, y):
 """
 Adds x and y. For example:

 >>> add(2, 2)
 5
 >>>
 """
 return x + y
```

- Useful for making sure help documentation matches the implementation

## Using doctest

- Run it as a command line tool on your code

```
bash % python -m doctest simple.py
```

- It even works on documentation files

```
bash % python -m doctest README.txt
```

- Anything that might have interactive Python sessions shown inside

# Using doctest

- Test failures produce a report

```
bash % python -m doctest simple.py

File "simple.py", line 6, in simple.add
Failed example:
 add(2, 2)
Expected:
 5
Got:
 4

1 items had failures:
 1 of 1 in simple.add
Test Failed 1 failures.
```

# Doctest Caution

- Don't go overboard with doctest
- Only use it as a documentation sanity check
- Do not use it for exhaustive testing
- Overuse is often a source of problems  
(output of Python might change over time)

# unittest Module

- A more formal testing framework
- Better suited for exhaustive testing
- Can deal with more complex test cases

# Using unittest

- First, you create a separate file

```
testsimple.py
import simple
import unittest
```

- Then you define testing classes

```
class TestAdd(unittest.TestCase):
 ...
```

- They must inherit from `unittest.TestCase`

# Using unittest

- Define testing methods

```
class TestAdd(unittest.TestCase):
 def test_simple(self):
 # Test with simple integer arguments
 r = simple.add(2, 2)
 self.assertEqual(r, 5)

 def test_str(self):
 # Test with strings
 r = simple.add('hello', 'world')
 self.assertEqual(r, 'helloworld')
```

- Each method must start with "test..."

# Using unittest

- Each test uses special assertions

```
Assert that expr is True
self.assertTrue(expr)

Assert that x == y
self.assertEqual(x,y)

Assert that x != y
self.assertNotEqual(x,y) # Assert x != y

Assert that x is near y
self.assertAlmostEqual(x,y,places)

Assert that callable(arg1,arg2,...) raises exc
self.assertRaises(exc,callable,arg1,arg2,...)
```

- There are others

# Running unittests

- To run tests, add the following code

```
testsimple.py
...
if __name__ == '__main__':
 unittest.main()
```

- Then run Python on the test file

```
bash % python testsimple.py
F.
=====
FAIL: test_simple (__main__.TestAdd)

Traceback (most recent call last):
 File "testsimple.py", line 8, in test_simple
 self.assertEqual(r, 5)
AssertionError: 4 != 5

Ran 2 tests in 0.000s
FAILED (failures=1)
```

## unittest comments

- There is an art to effective unit testing
- Can grow to be quite complicated for large applications
- The unittest module has a huge number of options related to test runners, collection of results, and other aspects of testing (consult documentation for details)

# Third Party Test Tools

- nose - Test discovery

<https://nose.readthedocs.org>

- coverage - Code coverage

<http://nedbatchelder.com/code/coverage/>

- mock - Mocking and testing library

<http://www.voidspace.org.uk/python/mock/>

## Exercise 7.1

Time : 15 Minutes



# logging Module

- A commonly used module for recording diagnostic information
- It's also a very large module with a lot of sophisticated functionality
- Will show a simple example to illustrate

# Exceptions Revisited

- In the exercises, we wrote a function `parse()` that looked something like this:

```
fileparse.py
def parse(f, types=None, names=None, delimiter=None):
 records = []
 for line in f:
 line = line.strip()
 if not line: continue
 try:
 records.append(split(line, types, names, delimiter))
 except ValueError as e:
 print "Couldn't parse :", line
 print "Reason : %s" % e
 return records
```

- Now, focus on the try-except section

# Exceptions Revisited

- Do you print a warning message?

```
try:
 records.append(split(line,types,names,delimiter))
except ValueError as e:
 print "Couldn't parse :", line
 print "Reason : %s" % e
```

- Or, do you silently ignore?

```
try:
 records.append(split(line,types,names,delimiter))
except ValueError as e:
 pass
```

- Neither solution is satisfactory because you often want both behaviors (user selectable)

# Using Logging

- The logging module can address this

```
fileparse.py
import logging
log = logging.getLogger(__name__)

def parse(f,types=None,names=None,delimiter=None):
 ...
 try:
 records.append(split(line,types,names,delimiter))
 except ValueError as e:
 log.warning("Couldn't parse : %s", line)
 log.debug("Reason : %s", e)
```

- Here, code is modified to issue warning messages on a special "Logger" object

# Logging Basics

- Creating a logger object

```
log = logging.getLogger(name) # name is a string
```

- Issuing log messages

```
log.critical(message [, args])
log.error(message [, args])
log.warning(message [, args])
log.info(message [, args])
log.debug(message [, args])
```

each method  
represents a different  
level of severity

- Each of the above methods creates a formatted log message (args is used for % operator)

```
logmsg = message % args # Written to the log
```

# Logging Configuration

- Logging behavior is configured separately

```
main.py
...
if __name__ == '__main__':
 import logging
 logging.basicConfig(
 filename = "app.log", # Log output file
 level = logging.INFO, # Output level
)
```

- Typically, this is a one-time configuration at program startup
- Separate from code that makes logging calls

# Big Picture

- Logging is highly configurable
- Can adjust every aspect of it (output files, levels, message formats, etc.)
- Code that uses logging doesn't have to worry about that however (it just issues messages)
- See: [practical-python/Optional/Logging.pdf](http://practical-python.com/docs/Optional/Logging.pdf)

## Exercise 7.2

Time : 10 Minutes

# Assertions

- assert statement

```
assert expr [, "diagnostic message"]
```

- If expression is not true, raises `AssertionError` exception
- Should not be used to check user-input
- Use for internal program checking

# Contract Programming

- Consider assertions on all inputs and outputs

```
def add(x, y):
 assert isinstance(x, int), "Expected int"
 assert isinstance(y, int), "Expected int"
 return x + y
```

- Checking inputs will immediately catch callers who aren't using appropriate arguments

```
>>> add(2, 3)
5
>>> add("2", "3")
Traceback (most recent call last):
...
AssertionError: Expected int
>>>
```

# Optimized mode

- Python has an optimized run mode

```
bash % python -O prog.py
```

- This strips all assert statements
- Allows debug/release mode development
- Normal mode for full debugging
- Optimized mode for faster production runs

## \_\_debug\_\_ variable

- Global variable checked for debugging

```
if __debug__:
 # Perform some kind of debugging code
 ...
```

- By default, `__debug__` is True
- Set False in optimized mode (`python -O`)
- The implementation is efficient. The if statement is stripped in both cases and in `-O` mode, the debugging code is stripped entirely.

# Error Handling

- Keeping Python alive upon termination

```
bash % python -i blah.py
Traceback (most recent call last):
 File "blah.py", line 13, in ?
 foo()
 File "blah.py", line 10, in foo
 bar()
 File "blah.py", line 7, in bar
 spam()
 File "blah.py", line 4, in spam
 x.append(3)
AttributeError: 'int' object has no attribute 'append'
>>>
```

- Python enters normal interactive mode
- Can use to examine global data, objects, etc.

# The Python Debugger

- pdb module
- Entering the debugger after a crash

```
bash % python -i blah.py
Traceback (most recent call last):
 File "blah.py", line 13, in ?
 foo()
 File "blah.py", line 10, in foo
 bar()
 File "blah.py", line 7, in bar
 spam()
 File "blah.py", line 4, in spam
 x.append(3)
AttributeError: 'int' object has no attribute 'append'
>>> import pdb
>>> pdb.pm()
> /Users/beazley/Teaching/blah.py(4)spam()
-> x.append(3)
(Pdb)
```

# The Python Debugger

- Launching the debugger inside a program

```
def some_function():
 statements
 ...
 import pdb; pdb.set_trace() # Enter the debugger
 ...
 statements
```

- This starts the debugger at the point of the `set_trace()` call

# Python Debugger

- Common debugger commands

```
(Pdb) help # Get help
(Pdb) w(here) # Print stack trace
(Pdb) d(own) # Move down one stack level
(Pdb) u(p) # Move up one stack level
(Pdb) b(reak) loc # Set a breakpoint
(Pdb) s(tep) # Execute one instruction
(Pdb) c(ontinue) # Continue execution
(Pdb) l(ist) # List source code
(Pdb) a(rgs) # Print args of current function
(Pdb) !statement # Execute statement
```

- For breakpoints, location is one of

```
(Pdb) b 45 # Line 45 in current file
(Pdb) b file.py:45 # Line 34 in file.py
(Pdb) b foo # Function foo() in current file
(Pdb) b module.foo # Function foo() in a module
```



# Python Debugger

- Running entire program under debugger

```
bash % python -m pdb someprogram.py
```

- Automatically enters the debugger before the first statement (allowing you to set breakpoints and change the configuration)

# Remote Debugging

- Use rpdb (<https://pypi.python.org/pypi/rpdb/>)

```
def some_function():
 statements
 ...
 import rpdb; rpdb.set_trace()
 ...
 statements
```

- Stops and allows pdb access via telnet

```
bash % telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
> /some/code/program.py(10)some_function()
-> statement
(Pdb)
```

# Profiling

- cProfile module
- Collects statistics and prints a report
- Run it from the command shell

```
bash % python -m cProfile someprogram.py
```

## Profile Sample Output

```
bash % python -m cProfile cparse.py
447981 function calls (446195 primitive calls) in
5.640 CPU seconds
```

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno
(function)					
2	0.000	0.000	0.000	0.000	:0(StringIO)
101599	0.470	0.000	0.470	0.000	:0(append)
56	0.000	0.000	0.000	0.000	:0(callable)
4	0.000	0.000	0.000	0.000	:0(close)
1028	0.010	0.000	0.010	0.000	:0(cmp)
4	0.000	0.000	0.000	0.000	:0(compile)
1	0.000	0.000	0.000	0.000	:0(digest)
2	0.000	0.000	0.000	0.000	:0(exc_info)
1	0.000	0.000	5.640	5.640	:0(execfile)
4	0.000	0.000	0.000	0.000	:0(extend)
50	0.000	0.000	0.000	0.000	:0(find)
83102	0.430	0.000	0.430	0.000	:0(get)

...

# Summary

- Testing with doctest and unittest
- Logging
- Debugging (pdb)
- Profiling

## Exercise 7.3

Time : 10 Minutes

## Section 8

# Generators

## Iteration

- A simple definition: Looping over items

```
a = [2,4,10,37,62]
Iterate over a
for x in a:
 ...
```

- A very common pattern
- loops, list comprehensions, etc.
- Most programs do a huge amount of iteration

# Iteration Everywhere

- Many different objects support iteration

```
a = "hello"
for c in a: # Loop over characters in a
 ...

b = { 'name': 'Dave', 'password':'foo'}
for k in b: # Loop over keys in dictionary
 ...

c = [1,2,3,4]
for i in c: # Loop over items in a list/tuple
 ...

f = open("foo.txt")
for x in f: # Loop over lines in a file
 ...
```

# Iteration: Protocol

- An inside look at the for statement

```
for x in obj:
 # statements
```

- Underneath the covers

```
_iter = obj.__iter__() # Get iterator object
while True:
 try:
 x = _iter.next() # Get next item
 except StopIteration: # No more items
 break
 # statements
 ...
```

- Objects that work with the for-loop all implement this low-level iteration protocol

# Iteration: Protocol

- Example: Manual iteration over a list

```
>>> x = [1,2,3]
>>> it = x.__iter__()
>>> it
<listiterator object at 0x590b0>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
>>> it.next()
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
StopIteration
>>>
```

# Supporting Iteration

- Knowing about iteration is useful if you want to add it to your own objects
- Example: Custom containers

```
class Portfolio(object):
 def __init__(self):
 self.holdings = []
 def __iter__(self):
 return self.holdings.__iter__()
 ...

port = Portfolio()
for s in port:
 ...
```

# The itertools Module

- A library module with various functions designed to help with iteration

```
itertools.chain(s1,s2)
itertools.count(n)
itertools.cycle(s)
itertools.dropwhile(predicate, s)
itertools.groupby(s)
itertools.ifilter(predicate, s)
itertools.imap(function, s1, ... sN)
itertools.repeat(s, n)
itertools.tee(s, ncopies)
itertools.izip(s1, ... , sN)
```

- All functions process data iteratively.
- Implement various kinds of iteration patterns

## Exercise 8.1

Time : 10 Minutes

# Customizing Iteration

- Suppose you wanted to create your own custom iteration pattern
- Example: Counting down...

```
>>> for x in countdown(10):
... print x,
...
10 9 8 7 6 5 4 3 2 1
>>>
```

- It turns out there is a very easy way to do it

# Generators

- A function that defines iteration

```
def countdown(n):
 while n > 0:
 yield n
 n -= 1
>>> for i in countdown(5):
... print i,
...
5 4 3 2 1
>>>
```

- Any function that uses yield is a generator



# Generator Functions

- Behavior is totally different than normal func
- Calling a generator function creates an generator object. It does not start running the function.

```
def countdown(n):
 print "Counting down from", n
 while n > 0:
 yield n
 n -= 1
```

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>>
```

Notice that no  
output was  
produced

# Generator Functions

- Function only executes on next()

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>> x.next()
Counting down from 10
10
>>>
```

Function starts  
executing here

- yield produces a value, but suspends function
- Function resumes on next call to next()

```
>>> x.next()
9
>>> x.next()
8
>>>
```

# Generator Functions

- When the generator returns, iteration stops

```
>>> x.next()
1
>>> x.next()
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
StopIteration
>>>
```

- Observation :A generator function implements the same low-level protocol that the for statement uses on lists, tuples, dicts, files, etc.

## Exercise 8.2

Time : 15 Minutes

# Producers & Consumers


- Generators are closely related to various forms of "producer-consumer" programming

producer

```
def follow(f):
 ...
 while True:
 ...
 yield line
 ...
```

consumer

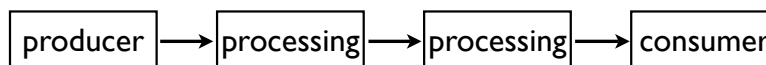
```
for line in follow(f):
 ...
```



- yield produces values
- for consume values

# Generator Pipelines

- You can use this aspect of generators to set up processing pipelines (like Unix pipes)
- Big picture:



- Processing pipes have an initial data producer, some set of intermediate processing stages, and a final consumer

# Generator Pipelines



```
def producer():
 ...
 yield item
 ...
```

- Producer is typically a generator (although it could also be a list or some other sequence)
- yield feeds data into the pipeline

Copyright (C) 2014, <http://www.dabeaz.com>

8- 17

# Generator Pipelines



```
def producer():
 ...
 yield item
 ...
```

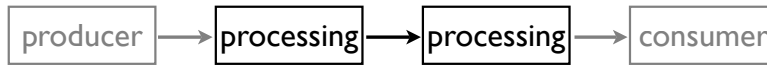
```
def consumer(s):
 for item in s:
 ...
```

- Consumer is just a simple for-loop
- It gets items and does something with them

Copyright (C) 2014, <http://www.dabeaz.com>

8- 18

# Generator Pipelines



```
def producer():
 ...
 yield item
 ...

def processing(s):
 for item in s:
 ...
 yield newitem
 ...

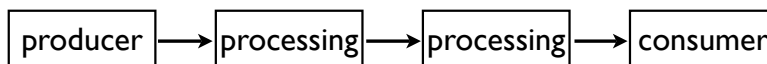
def consumer(s):
 for item in s:
 ...
```

- Intermediate processing stages simultaneously consume and produce items
- They might modify the data stream
- They can also filter (discarding items)

Copyright (C) 2014, <http://www.dabeaz.com>

8- 19

# Generator Pipelines



```
def producer():
 ...
 yield item
 ...

def processing(s):
 for item in s:
 ...
 yield newitem
 ...

def consumer(s):
 for item in s:
 ...
```

Arrows in the original image point from 'yield item' to the first 'processing' function and from 'yield newitem' to the 'consumer' function.

- Code to set up the pipeline

```
a = producer()
b = processing(a)
c = consumer(b)
```

Arrows in the original image point from 'a' to 'processing(a)' and from 'b' to 'consumer(b)'.

- You will notice that data incrementally flows through the different functions

Copyright (C) 2014, <http://www.dabeaz.com>

8- 20

# Exercise 8.3

Time : 15 minutes

## Generator Expressions

- A generator version of a list comprehension

```
>>> a = [1,2,3,4]
>>> b = (2*x for x in a)
>>> b
<generator object at 0x58760>
>>> for i in b: print i,
...
2 4 6 8
>>>
```

- Important differences
  - Does not construct a list.
  - Only useful purpose is iteration
  - Once consumed, can't be reused

# Generator Expressions

- General syntax

```
(expression for i in s if conditional)
```

- Can also serve as a function argument

```
sum(x*x for x in a)
```

- Can be applied to any iterable

```
>>> a = [1,2,3,4]
>>> b = (x*x for x in a)
>>> c = (-x for x in b)
>>> for i in c: print i,
...
-1 -4 -9 -16
>>>
```

# Generator Expressions

- Main use of generator expressions is in code that performs some calculation on a sequence, but only uses the result once
- Example : Strip all comments from a file

```
f = open("somefile.txt")
lines = (line for line in f if not line.startswith("#"))
for line in lines:
 ...
f.close()
```

- Code runs faster and uses little memory (it's like a filter applied to a stream)

# Exercise 8.4

Time : 10 Minutes

## Why Use Generators?

- Many problems are much more clearly expressed in terms of iteration
- Looping over a collection of items and performing some kind of operation (searching, replacing, modifying, etc.)
- Processing pipelines can be applied to a wide range of data processing problems



# Why Use Generators?

- Generators encourage code reuse
- Separate the "iteration" from code that uses the iteration.
- Means that various iteration patterns can be defined more generally.

# Why Use Generators?

- Better memory efficiency
- "Lazy" evaluation
- Only produce values when needed
- Contrast to constructing a big list of values first
- Can operate on infinite data streams

# The itertools Module

- A library module with various functions designed to help with iterators/generators

```
itertools.chain(s1,s2)
itertools.count(n)
itertools.cycle(s)
itertools.dropwhile(predicate, s)
itertools.groupby(s)
itertools.ifilter(predicate, s)
itertools.imap(function, s1, ... sN)
itertools.repeat(s, n)
itertools.tee(s, ncopies)
itertools.izip(s1, ... , sN)
```

- All functions process data iteratively.
- Implement various kinds of iteration patterns

## More Information

- "Generator Tricks for Systems Programmers" tutorial from PyCon'08

<http://www.dabeaz.com/generators>

- More examples and more generator tricks

Section 9 (Optional)

# Text I/O Handling

## Overview

- This sections expands upon text processing
- Generating text
- Text I/O
- Unicode

# Generating Text

- Programs often need to generate text
- Reports
- HTML pages
- XML
- Endless possibilities

# String Concatenation

- Strings can be concatenated using +

```
s = "Hello"
t = "World"

a = s + t # a = "HelloWorld"
```

- Although (+) is fine for just a few strings, it has horrible performance if you are concatenating many small chunks together to create a large string
- Should not be used for generating output

# String Joining

- The fastest way to join many strings

```
chunks = ["chunk1", "chunk2", ... "chunkN"]
result = separator.join(chunks)
```

- Example:

```
chunks = ["Is", "Chicago", "Not", "Chicago?"]

" ".join(chunks) → "Is Chicago Not Chicago?"
",".join(chunks) → "Is,Chicago,Not,Chicago?"
"".join(chunks) → "IsChicagoNotChicago?"
```

# String Joining Example

- Don't do this:

```
s = ""
for x in seq:
 ...
 s += "some text being produced"
 ...
```

- Better:

```
chunks = []
for x in seq:
 ...
 chunks.append("some text being produced")
 ...
s = "".join(chunks)
```

# String Interpolation

- In languages like Perl and Ruby, programmers are used to string interpolation features

```
$name = "Dave";
$age = 39;

print "$name is $age years old\n";
```

- Python doesn't have a direct equivalent
- However, there are some alternatives

# Built-in Formatting

- Use the `format()` method

```
print "{name} is {age} years old".format(name="Dave",age=39)
```

- Use the `%` operator with a dictionary

```
fields = {
 'name' : 'Dave',
 'age' : 39
}

print "%(name)s is %(age)s years old" % fields
```

# Template Strings

- A special string that supports \$substitutions

```
import string
s = string.Template("$name is $age years old\n")

print s.substitute(name='Dave',age=39)
```

- Or you can supply a dictionary

```
fields = {
 'name' : 'Dave',
 'age' : 39
}
print s.substitute(fields)
```

- To ignore missing values, use this alternative

```
s.safe_substitute(fields)
```

## Exercise 9.1

Time : 15 Minutes

# Text Input/Output

- You frequently read/write text from files
- Example: Reading line-by-line

```
f = open("something.txt", "r")
for line in f:
 ...
```

- Example: Writing a line of text

```
f = open("something.txt", "w")
f.write("Hello World\n")

print >>f, "Hello World\n"
```

- There are still a few issues to worry about

# Line Handling

- Question: What is a text line?
- It's different on different operating systems

```
some characters \n (Unix)
some characters \r\n (Windows)
some characters \r (Mac)
```

- By default, Python uses the system's native line ending when writing text files
- However, it can get messy when reading text files (especially cross platform)



# Line Handling

- Example: Reading a Windows text file on Unix

```
>>> f = open("test.txt", "r")
>>> f.readlines()
['Hello\r\n', 'World\r\n']
>>>
```

- Notice how the lines include the extra Windows '\r' character
- This is a potential source of problems for programs that only expect '\n' line endings

# Universal Newline

- Python has a special "Universal Newline" mode

```
>>> f = open("test.txt", "U")
>>> f.read()
'Hello World\n'
>>>
```

- Converts all endings to standard '\n' character
- f.newlines records the actual newline character that was used in the file

```
>>> f.newlines
'\r\n'
>>>
```

# Universal Newline

- Example: Reading a Windows text file on Unix

```
>>> f = open("test.txt","r")
>>> f.readlines()
['Hello\r\n', 'World\r\n']
```

```
>>> f = open("test.txt","U")
>>> f.readlines()
['Hello\n', 'World\n']
>>> f.newlines
'\r\n'
>>>
```

- Notice how non-native Windows newline '\r\n' is translated to standard '\n'

# Text Encoding

- Question :What is a character?
- In Python 2, text consists of 8-bit characters

"Hello World" → 48 65 6c 6c 6f 20 57 6f 72 6c 64

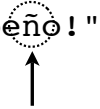
- Characters are usually encoded in ASCII

	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	\$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074

# International Characters

- Problem : How to deal with characters from international character sets?

"That's a spicy Jalapeño!"



- Question: What is the character encoding?
- Historically, everyone made a different encoding

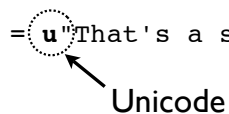
ñ	=	0x96	(MacRoman)
ñ	=	0xf1	(CP1252 - Windows)
ñ	=	0xa4	(CP437 - DOS)

- Bloody hell!

# Unicode

- For international characters, use Unicode
- In Python, there is a special syntax for literals

t = "That's a spicy Jalapeño!"



- Unicode strings are just like regular strings except that they hold Unicode characters
- What is a Unicode character?

# Unicode Characters

- Unicode defines a standard numerical value for every character used in all languages (except for fictional ones such as Klingon)
- The numeric value is known as "code point"
- There are a lot of code points (> 100,000)

ñ = U+00F1  
ε = U+03B5  
ϣ = U+0A87  
イテ = U+3304

# Unicode Charts

- <http://www.unicode.org/charts>

European Alphabets	African Scripts	Indic Scripts	East Asian Scripts	Central Asian Scripts
(see also Comb. Marks)	<b>Ethiopic</b>	Bengali	<b>Han Ideographs</b>	Kharoshthi
<b>Armenian</b>	Ethiopic	Devanagari	Unified CJK Ideographs (5MB)	Mongolian
Armenian	Ethiopic Supplement	Gujarati	CJK Ideographs Ext. A (2MB)	Phags-Pa
<b>Armenian Ligatures</b>	Ethiopic Extended	Gurmukhi	CJK Ideographs Ext. B (13MB)	Tibetan
<b>Coptic</b>	<b>Other African scripts</b>	Kannada		<b>Ancient Scripts</b>
Coptic	N'Ko	Lepcha	Compatibility Ideographs (.5MB)	<b>Ancient Greek</b>

# Using Unicode Charts

0080

C1 Controls and Latin-1 Supplement

00FF

	008	009	00A	00B	00C	00D	00E	00F
0	☐ 0080	☐ 0090	NB SP 00A0	◦ 00B0	À 00C0	Đ 00D0	à 00E0	ð 00F0
1	☐ 0081	☐ 0091	¡ 00A1	± 00B1	Á 00C1	Ñ 00D1	á 00E1	ñ 00F1

t = u" That's a spicy Jalape\u00f1o!"

- \uxxxx - Embeds a Unicode code point in a string
- Code points specified in hex by convention

Copyright (C) 2014, <http://www.dabeaz.com>

9- 21

# Using Unicode Charts

- All code points also have descriptive names

```
00F1 ñ LATIN SMALL LETTER N WITH TILDE
 ≡ 006E n 0303 õ
00F2 ò LATIN SMALL LETTER O WITH GRAVE
 ≡ 006F o 0300 ò
00F3 ó LATIN SMALL LETTER O WITH ACUTE
 ≡ 006F o 0301 ó
```

- \N{name} - Embeds a named character

t = u"Spicy Jalape\N{LATIN SMALL LETTER N WITH TILDE}o!"

Copyright (C) 2014, <http://www.dabeaz.com>

9- 22

# Unicode Representation

- Internally, Unicode chars are 16 or 32 bits

```
t = u"Jalape\u00f1o"
```

```
004a 0061 006c 0061 0070 0065 00f1 006f
```

- Normally, you don't worry about this
- Except you have to perform I/O

```
u'J' --> 00 4a (Big Endian)
u'J' --> 4a 00 (Little Endian)
```

- How do characters get encoded in the file?

# Unicode I/O

- Unicode does not define a standard file encoding--it only defines character code values
- There are many different file encodings
- Examples: UTF-8, UTF-16, etc.
- Most popular: UTF-8 (ASCII is a subset)
- So, how do you deal with these encodings?

# Unicode File I/O

- Unicode I/O handled using io module
- `io.open(filename,mode,encoding="enc")`

```
>>> f = io.open("data.txt", "w", encoding="utf-8")
>>> f.write(u"Hello World\n")
>>> f.close()

>>> f = io.open("data.txt", "w", encoding="utf-16")
>>> f.write(data)
>>>
```

- Several hundred encodings are supported
- Compatibility note : In older Python versions, Unicode I/O handled by 'codecs' module

# Unicode Encoding

- Explicit encoding to bytes

```
>>> a = u"Jalape\u00f1o"
>>> enc_a = a.encode("utf-8")
>>>
```

- Explicit decoding from bytes

```
>>> enc_a = 'Jalape\xc3\xb1o'
>>> a = enc_a.decode("utf-8")
>>> a
u'Jalape\x1f1o'
>>>
```

# Encoding Errors

- Encoding/Decoding text is often messy
- May encounter broken/invalid data
- The default behavior is to raise an `UnicodeError` Exception

```
>>> a = u"Jalape\xf1o"
>>> b = a.encode("ascii")
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character
u'\xf1' in position 6: ordinal not in range(128)
>>>
```

# Encoding Errors

- Encoding/Decoding can use an alternative error handling policy

```
s.decode("encoding", errors)
s.encode("encoding", errors)
```

- Errors is one of

'strict'	Raise exception (the default)
'ignore'	Ignore errors
'replace'	Replace with replacement character
'backslashreplace'	Use escape code
'xmlcharrefreplace'	Use XML character reference



# Encoding Errors

- Example: Ignore bad characters

```
>>> a = u"Jalape\xf1o"
>>> a.encode("ascii", 'ignore')
'Jalapeo'
>>>
```

- Example: Encode Unicode into ASCII

```
>>> a = u"Jalape\xf1o"
>>> b = a.encode("us-ascii", "xmlcharrefreplace")
'Jalapeño'
>>>
```

# Finding the Encoding

- How do you determine the encoding of a file?
- Might be known in advance (in the manual)
- May be indicated in the file itself

```
<meta http-equiv="Content-Type"
 content="text/html; charset=UTF-8" />
```

- Depends on the data source, application, etc.

# Unicode Everywhere

- Unicode is the modern standard for text
- In Python 3, all text is Unicode
- Here are some basic rules to remember:
  - All text files are encoded (even ASCII)
  - When you read text, you always decode
  - When you write text, you always encode

## A Caution

- Unicode may sneak in when you don't expect it
  - Database integration
  - XML Parsing
- Unicode silently propagates through string-ops

```
s = "Spicy" # Standard 8-bit string
t = u"Jalape\u00f1o" # Unicode string

w = s + t # Unicode : u'SpicyJalape\u00f1o'
```

- This propagation may break your code if it's not expecting to receive Unicode text

# Exercise 9.2

Time : 10 Minutes

## Section 10 (Optional)

# A Few Advanced Topics

## Overview

- More Python features you may encounter
  - Variable argument functions
  - Anonymous functions and lambda
  - Closures
  - Function decorators
  - Static and class methods
  - Properties
  - Packages

# Variable Arguments

- Function that accepts any number of args

```
def foo(x, *args):
 ...
```

- Here, the arguments get passed as a tuple

```
foo(1, 2, 3, 4, 5)
 ↓
def foo(x, *args):
 ↓ ↓
 1 (2, 3, 4, 5)
```

# Variable Arguments

- Function that accepts any keyword args

```
def foo(x, y, **kwargs):
 ...
```

- Extra keywords get passed in a dict

```
foo(2, 3, flag=True, mode="fast", header="debug")
 ↓ ↓ ↓
def foo(x, y, **kwargs):
 ...
 ↓
 { 'flag' : True,
 'mode' : 'fast',
 'header' : 'debug' }
```

# Variable Arguments

- A function that takes any arguments

```
def foo(*args,**kwargs):
 statements
```

- This will accept any combination of positional or keyword arguments
- Sometimes used when writing wrappers or when you want to pass arguments through to another function

# Passing Tuples and Dicts

- Tuples can be expand into function args

```
args = (2,3,4)
foo(1, *args) # Same as foo(1,2,3,4)
```

- Dictionaries can expand to keyword args

```
kwargs = {
 'color' : 'red',
 'delimiter' : ',',
 'width' : 400 }

foo(data, **kwargs)
Same as foo(data,color='red',delimiter=',',width=400)
```

- These are not commonly used except when writing library functions.

# Exercise 10.1

Time : 10 Minutes

## List Sorting Revisited

- Lists can be sorted "in-place" (sort method)

```
s = [10,1,7,3]
s.sort() # s = [1,3,7,10]
```

- Sorting in reverse order

```
s = [10,1,7,3]
s.sort(reverse=True) # s = [10,7,3,1]
```

- It seems "simple" enough...

# List Sorting

- Sometimes you need to perform extra processing while sorting
- Example: Case-insensitive string sort

```
>>> s = ["hello", "WORLD", "test"]
>>> s.sort()
>>> s
['WORLD', 'hello', 'test']
>>>
```

- Here, we might like to fix the order

# List Sorting

- You can fix this using a "key function"

```
>>> def tolower(x):
... return x.lower()
...
>>> s = ["hello", "WORLD", "test"]
>>> s.sort(key=tolower)
>>> s
['hello', 'test', 'WORLD']
>>>
```

- The key function is a "callback function" that the `sort()` method applies to each item
- The value returned by the key function determines the sort order



# Callback Functions

- Callback functions are often short one-line functions that are only used for that one operation (e.g., sorting)
- Programmers often ask for a short-cut
- For example, is there some shorter way to specify the custom processing for `sort()`?

# Anonymous Functions

- lambda expression
- Creates an unnamed function that evaluates a single expression
- The above code is a shorter version of this

```
names.sort(key=lambda s: s.lower())
```

```
Same as
def lowerkey(s):
 return s.lower()

names.sort(key=lowerkey)
```

# Using lambda

- lambda is highly restricted
- Only a single expression is allowed and you can't use statements such as if, while, print, for, etc.
- Most common use is with sort()
- Sometimes seen with map() and filter() functions which carry out the same work as a list comprehension (more modern)

# lambda and map()

- Legacy code example

```
>>> nums = [1,2,3,4]
>>> squares = map(lambda x: x*x, nums)
>>> squares
[1, 4, 9, 16]
>>>
```

- Modern implementation

```
>>> nums = [1,2,3,4]
>>> squares = [x*x for x in nums]
>>> squares
[1, 4, 9, 16]
>>>
```

- List comprehension runs 1.5-2x faster

# Advice on Lambda

- Use lambda sparingly
- It's never necessary to use it and going overboard is a good way to create a program that's hard to figure out later
- Replace map() and filter() calls in old code with list comprehensions
- Most popular use : with sorting

## Exercise 10.2

Time : 5 Minutes

# Returning Functions

- Consider the following function

```
def add(x,y):
 def do_add():
 print "Adding",x,y
 return x+y
 return do_add
```

- A function that returns another function?

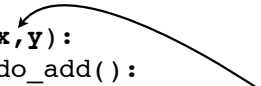
```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
Adding 3 4
7
```

- Notice that it works, but ponder it...

# Local Variables

- Observe how the inner function refers to variables defined by the outer function

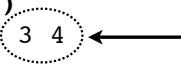
```
def add(x,y):
 def do_add():
 print "Adding",x,y
 return x+y
 return do_add
```



- Further observe that those variables are somehow kept alive after add() has finished

```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
Adding: 3 4
7
```

← Where are these values coming from?



# Closures

- If an inner function is returned as a result, the inner function is known as a "closure"

```
def add(x,y):
 def do_add():
 print "Adding",x,y
 return x+y
 return do_add
```

- Essential feature :A "closure" retains the values of all variables needed for the function to run properly later on

# Closures

- To make it work, references to the outer variables (free variables) get carried along with the function

```
>>> a
<function do_add at 0x4dd30>
>>> a.__closure__
(<cell at 0x54f30: int object at 0x54fe0>,
 <cell at 0x54fd0: int object at 0x54f60>)
>>> a.__closure__[0].cell_contents
4
>>> a.__closure__[1].cell_contents
3
```

- So, think of a closure as a function, but with an extra environment of variable definitions that's sitting behind the scenes

# Using Closures

- Closures are an essential feature of Python
- However, their use is often subtle
- Common applications:
  - Use in callback functions
  - Delayed evaluation
  - Decorator functions (later)

# Delayed Evaluation

- Go back to our original example
- This is an example of "delayed evaluation"
- `add()` doesn't do anything, it returns a function that carries out work later

```
def add(x,y):
 def do_add():
 print "Adding",x,y
 return x+y
 return do_add

>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
Adding 3 4
7
```

# Delayed Evaluation

- Delayed evaluation is sometimes used to defer expensive calculations until later
- Perhaps the calculation won't be needed after all (so if you throw it away, you haven't paid a big penalty for doing the work)
- Also sometimes used as a trick to avoid excessive code repetition

## Exercise 10.3

Time : 15 Minutes

# Function Decorators

- Closures are the basis for another Python feature known as "decoration"
- A topic related to "metaprogramming"
- An advanced topic that we'll only scratch the surface of right now

## An Example

- Consider a function

```
def add(x, y):
 return x + y
```

- Now, consider the function with some logging

```
def add(x, y):
 print "Calling add"
 return x + y
```

- Now, a second function with some logging

```
def sub(x, y):
 print "Calling sub"
 return x - y
```

- Observation: It's kind of repetitive



# Observation

- Writing programs where there is a lot of code replication is usually really annoying
- Tedious to write
- Hard to maintain
- Especially if you decide that you want to change how it works (i.e., a different kind of logging perhaps).

# An Example

- Perhaps you can make logging wrappers

```
def logged(func):
 def wrapper(*args, **kwargs):
 print 'Calling', func.__name__
 return func(*args, **kwargs)
 return wrapper
```

- Now, consider this bit of code

```
def add(x, y):
 return x + y

add = logged(add)
```

- Question: What happens here?

```
add(3, 4)
```

# An Example

- This example illustrates the process of creating a so-called "wrapper function"
- A wrapper is a function that wraps another function with some extra bits of processing

```
>>> add(3, 4)
Calling add ← extra output
7 (added by the wrapper)
>>>
```

- Notice: the `logged()` function creates the wrapper and returns it as a result

# Decorators

- Putting wrappers around functions is extremely common in Python

```
def add(x, y):
 return x + y
add = logged(add)
```

- So common, there is special syntax for doing it

```
@logged
def add(x, y):
 return x + y
```

- This performs the exact steps as shown at the top of the slide (it's just syntax)
- Is said to "decorate" the function

# Using Decorators

- Decorators are about as close as Python gets to having a macro system or a preprocessor
- A common use is to implement code that is either highly repetitive or so general purpose that it might be used by a large number of function definitions throughout an application
- Logging is just one example

# Commentary

- There are many more subtle details to decorators than what has been presented here
- For example, using them in classes
- Or using multiple decorators with a function
- However, the previous example is a good illustration of how their use tends to arise

# Exercise 10.4

Time : 10 Minutes

## Decorated Methods

- Predefined decorators are used to specify special kinds of methods in class definitions

```
class Foo(object):
 def bar(self,a):
 ...
 @staticmethod
 def spam(a):
 ...
 @classmethod
 def grok(cls,a):
 ...
 @property
 def name(self):
 ...
```

- Will briefly describe each one

# Static Methods

- `@staticmethod` is used to define a so-called "static" class methods (from C++/Java)
- A function that's part of the class, but which does not operate on instances

```
class Foo(object):
 @staticmethod
 def bar(x):
 print "x =", x
```

```
>>> Foo.bar(2)
x = 2
>>>
```

# Using Static Methods

- Sometimes used to implement internal supporting code for a class
- Example : Code to help manage created instances (memory management, system resources, persistence, locking, etc.)

# Class Methods

- `@classmethod` is used to define class methods
- A method that receives the class object as the first parameter instead of the instance

```
class Foo(object):
 def bar(self):
 print self
 @classmethod
 def spam(cls):
 print cls

>>> f = Foo()
>>> f.bar()
<__main__.Foo object at 0x971690> ← An instance
>>> Foo.spam()
<class '__main__.Foo'> ← A class
>>>
```

Copyright (C) 2014, <http://www.dabeaz.com>

10-37

# Using Class Methods

- Class methods are often used as a tool for defining alternate constructors

```
class Date(object):
 def __init__(self, year, month, day):
 self.year = year
 self.month = month
 self.day = day
 @classmethod
 def today(cls):
 tm = time.localtime()
 return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)

d = Date.today()
```

Notice how the class passed as an argument.

Copyright (C) 2014, <http://www.dabeaz.com>

10-38

# Using Class Methods

- Class methods solve some tricky problems with features like inheritance

```
class Date(object):
 ...
 @classmethod
 def today(cls):
 tm = time.localtime()
 return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)

class NewDate(Date):
 ...

d = NewDate.today()
```

Gets the correct class  
(e.g., NewDate)

## Exercise 10.5

Time : 10 Minutes

# Packages

- For larger collections of code, it is often desirable to organize modules into a hierarchy
- Example :A collection of graphics modules

```
graphics
 graphics.primitive
 graphics.primitive.line
 graphics.primitive.fill
 graphics.primitive.text
 graphics.twod
 graphics.twod.plot
 graphics.twod.contour
 graphics.formats
 graphics.formats.png
 graphics.formats.jpg
 ...
```

# Creating a Package

- First, organize source files as directory tree

```
graphics/
 primitive/
 line.py
 fill.py
 text.py
 twod/
 plot.py
 contour.py
 formats/
 png.py
 jpg.py
```

- This should mirror the package structure



# Creating a Package

- Next, add `__init__.py` files

```
graphics/
 __init__.py
 primitive/
 __init__.py
 line.py
 fill.py
 text.py
 twod/
 __init__.py
 plot.py
 contour.py
 formats/
 __init__.py
 png.py
 jpg.py
```

- These files can be empty

# Using a Package

- Importing a specific package component

```
import graphics.formats.jpg
f = graphics.formats.jpg.open("foo.jpg")
```

- During import, all `__init__.py` files execute

```
graphics/__init__.py
graphics/formats/__init__.py
graphics/formats/jpg.py
```

- Some import shortcuts

```
import graphics.formats.jpg as jpg
from graphics.formats import jpg

f = jpg.open("foo.jpg")
```

# \_\_init\_\_.py files

- The primary purpose of `__init__.py` is to supply code that must execute as parts of a package are imported

- Example: Automatically load submodules

```
graphics/formats/__init__.py

import jpeg, png, tiff, gif # Load submodules
```

- Example use:

```
import graphics.formats as formats
f = formats.jpeg.open("foo.jpeg")
g = formats.png.open("bar.png")
```

# Package Issues

- Using a component of a package is relatively straightforward (same as using a module)
- However, organizing code as a package introduces a number of subtle problems
- You'll need to consult a reference for the gory details

# Exercise 10.6

Time : 10 Minutes

# That's All Folks!

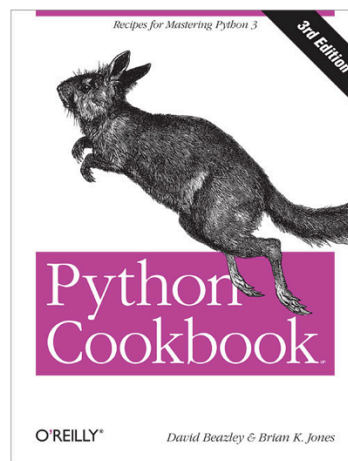
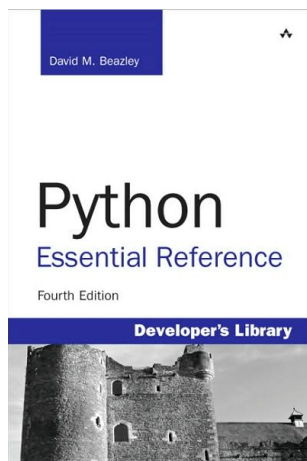
- End of the course
- Material covered should be enough to get you going with most aspects of Python development
- Much of your work is now going to involve features of various libraries and packages
- Hope you enjoyed the class!

# Advanced Topics

- Here are some major areas of Python that are covered in advanced classes
  - Network programming
  - Threads and concurrent programming
  - Distributed computing
  - Coroutines
  - Objects and metaprogramming

# Shameless Plug

- Buy my books!
- Twitter: @dabeaz



# Practical Python Index

## Symbols and Numbers

!= operator, 1-38  
#, program comment, 1-29  
%, floating point modulo operator, 1-53  
%, integer modulo operator, 1-49  
%, string formatting operator, 2-26, 2-29  
&, bit-wise and operator, 1-49  
&, set intersection operator, 2-22, 2-23  
() operator, 5-32  
(), tuple, 2-6  
\*\*, power operator, 1-53  
\*, list replication, 1-69  
\*, multiply operator, 1-49, 1-53  
\*, sequence replication operator, 2-32  
\*, string replication operator, 1-59  
+, add operator, 1-49, 1-53  
+, list concatenation, 1-67  
+, String concatenation, 1-58, 9-4  
-, set difference operator, 2-22, 2-23  
-, subtract operator, 1-49, 1-53  
-i option, Python interpreter, 7-29  
-O option, Python interpreter, 7-27, 7-28  
. operator, 5-32  
... interpreter prompt, 1-14  
/ division operator, 1-53  
/, division operator, 1-49  
//, floor division operator, 1-49, 1-50  
< operator, 1-38  
<<, left shift operator, 1-49  
<= operator, 1-38  
== operator, 1-38  
> operator, 1-38  
>= operator, 1-38  
>>, right shift operator, 1-49  
>>> interpreter prompt, 1-14

|, set union operator, 2-22, 2-23  
~, bit-wise negation operator, 1-49

## A

abs() function, 1-49, 1-53  
Absolute value function, 1-49, 1-53  
\_\_abs\_\_() method, 5-28  
Accessor methods, 6-35  
Accessor methods and properties, 6-43  
Adding items to dictionary, 2-13  
\_\_add\_\_() method, 5-28  
Advanced String Formatting, 9-8  
and operator, 1-38, 1-39  
\_\_and\_\_() method, 5-28  
anonymous function, lambda, 10-12  
append() method, of deque, 4-73  
append() method, of lists, 1-67  
appendleft() method, of deque, 4-73  
Argument naming style, 3-17  
Arguments, passing mutable objects, 3-25, 3-26  
argv variable, sys module, 4-26  
assert statement, 7-25  
assert statement, stripping in -O mode, 7-27  
assertEqual() method, unittest module, 7-12  
AssertionError exception, 7-25  
Assertions, and unit testing, 7-12  
assertNotEqual() method, unittest module, 7-12  
assertRaises() method, unittest module, 7-12  
assert\_() method, unittest module, 7-12  
assignment operations, 2-55  
Assignment, copying, 2-56, 2-58  
Assignment, reference counting, 2-56, 2-58  
Associative array, 2-12  
Attribute access functions, 5-35  
Attribute binding, 6-11  
Attribute lookup, 6-15, 6-18  
Attribute, definition of, 5-2, 5-3  
Attributes, computed using properties, 6-39, 6-42  
Attributes, modifying values, 6-12  
Attributes, private, 6-31, 6-32, 6-33  
Awk, and list comprehensions, 2-52

## B

Base class, 5-11  
\_\_bases\_\_ attribute of classes, 6-17  
Binding of attributes in objects, 6-11

- Block comments, 1-29
- Boolean expressions, 1-39
- Boolean type, 1-46, 1-47
- Booleans, and integers, 1-47
- Bottom up programming style, 3-10
- Bound method, 5-33, 5-34
- break statement, 2-38
- Breakpoint, debugger, 7-32
- Built-in exceptions, 3-34

## C

- C3 Linearization Algorithm, 6-20, 6-22, 6-23, 6-24, 6-25
- callback function, with sort(), 10-10
- Calling a function, 1-83
- Calling other methods in the same class, 5-9
- Case conversion, 1-60
- Catching exceptions, 1-86
- Catching multiple exceptions, 3-36
- Class implementation chart, 6-10
- class methods, 10-37
- class statement, 5-4
- class statement, defining methods, 5-8
- Class,, 6-8
- Class, representation of, 6-8
- class, static methods, 10-35, 10-36
- Class, `__slots__` attribute of, 6-44
- @classmethod decorator, 10-34, 10-37
- `__class__` attribute of instances, 6-9
- Closures, 10-19
- Code blocks and indentation, 1-34
- Code formatting, 1-43
- Code reuse, and generators, 8-27
- Colon, and indentation, 1-34
- Command line arguments, manual parsing, 4-27
- Command line options, 4-26
- Command line, running Python, 1-23
- Comments, 1-29
- Community links, 1-3
- Complex type, 1-46
- Computed attributes, 6-39, 6-42
- Concatenation of strings, 1-58
- Concatenation, lists, 1-67
- Concatenation, of sequences, 2-32
- Conditionals, 1-37
- Conditionals, on numpy arrays, 4-111
- Conformance checking of functions, 3-28
- Container, 2-16

- Containers, dictionary, 2-19, 2-20
- Containers, list, 2-17, 2-18
- context manager, 3-43
- continue statement, 2-39
- Contract programming, 7-26
- Conversion of numbers, 1-54
- Converting to strings, 1-63
- copy module, 2-62
- copy() function, shutil module, 4-36
- Copying and moving files, 4-36
- copying, lack of in assignment, 2-55
- copytree() function, shutil module, 4-36
- cos() function, math module, 1-53
- Counter objects, collections module, 4-74, 4-75, 4-76, 4-77, 4-78, 4-79, 4-80
- coverage, 7-15
- cPickle module, 4-72
- Creating new objects, 5-4
- Creating programs, 1-18
- ctime() function, time module, 4-34

## D

- Data structure, dictionary, 6-3
- Data structures, 2-5
- Database like queries on lists, 2-50
- date and time functions, 4-37
- Date and time manipulation, 4-38
- datetime module, 4-38
- Debugger, 7-30
- Debugger, breakpoint, 7-32
- Debugger, commands, 7-32
- Debugger, launching inside a program, 7-31
- Debugger, running at command line, 7-33, 7-34
- `__debug__` variable, 7-28
- decorators, 10-25
- Deep copies, 2-62
- deepcopy() function, copy module, 2-62
- def statement, 1-83, 3-7
- Defining a function, 3-7
- Defining new functions, 1-83
- Defining new objects, 5-4
- Definition order, 3-6
- del operator, lists, 1-70, 1-71
- del statement, 1-32
- delattr() function, 5-35
- Delayed evaluation,, 10-22
- Deleting items from dictionary, 2-13
- `__delitem__()` method, 5-29

- Derived class, 5-11
- Design by contract, 7-26
- Dictionary, 2-12
- Dictionary, and class representation, 6-8
- Dictionary, and module namespace, 6-4
- Dictionary, and object representation, 6-5
- Dictionary, testing for keys, 2-21
- Dictionary, updating and deleting, 2-13
- Dictionary, use as a container, 2-19, 2-20
- Dictionary, use as data structure, 6-3
- Dictionary, using as function keyword arguments, 10-6
- Dictionary, when to use, 2-14
- `__dict__` attribute, of instances, 6-6, 6-7
- directories, walking over a tree, 4-35
- Directory listing, 4-30
- distribute, 4-88
- `divmod()` function, 1-49, 1-53
- `__div__()` method, 5-28
- doctest module, 7-4, 7-5, 7-6
- doctest module, self-testing, 7-8
- Documentation, 1-16
- Documentation strings, and testing, 7-4, 7-5
- Double precision float, 1-51
- Double-quoted string, 1-55
- Downloads, 1-3

## E

- `easy_install` command, 4-88
- Edit,compile,debug cycle, 1-12
- Eggs, Python package format, 4-88
- `elif` statement, 1-37
- `else` statement, 1-37
- Embedded nulls in strings, 1-57
- empty code blocks, 1-42
- Enabling future features, 1-50
- Encapsulation, 6-28
- Encapsulation, and accessor methods, 6-35
- Encapsulation, and properties, 6-36
- Encapsulation, challenges of, 6-29
- Encapsulation, uniform access principle, 6-41
- `endswith()` method, strings, 1-61
- `enumerate()` function, 2-42, 2-43
- `environ` variable, `os` module, 4-29
- Environment variables, 4-29
- Error reporting strategy in exceptions, 3-39, 3-40, 3-41
- Escape codes, strings, 1-56, 1-64
- `except` statement, 1-86, 3-30
- Exception base class, 5-31

- Exception, defining new, 5-31
- Exceptions, 1-85, 3-30
- Exceptions, catching, 1-86
- Exceptions, catching any, 3-37
- Exceptions, catching multiple, 3-36
- Exceptions, caution on use, 3-38
- Exceptions, finally statement, 3-42
- Exceptions, how to report errors, 3-39, 3-40, 3-41
- Exceptions, ignoring, 3-37
- Exceptions, list of built-in, 3-34
- Exceptions, passed value, 3-35
- Exceptions, propagation of, 3-31, 3-32, 3-33
- Execution model, 1-28
- Execution of modules, 4-7
- `exists()` function, `os.path` module, 4-32, 4-33
- `exit()` function, `sys` module, 3-44
- Exploding heads, and exceptions, 3-38
- Exponential notation, 1-51
- Extended slicing of sequences, 2-34

## F

- False Value, 1-47
- File globbing, 4-30
- File system, copying and moving files, 4-36
- File system, getting a directory listing, 4-30
- File tests, 4-32, 4-33
- File, obtaining metadata, 4-34
- Files, and `for` statement, 1-77
- Files, and `with` statement, 1-80
- Files, end of file (EOF), 1-78, 1-79
- Files, opening, 1-76
- Files, reading in chunks, 1-78, 1-79
- Files, reading line by line, 1-77
- `__file__` attribute, of modules, 4-14, 4-15, 4-16, 4-17
- Filtering sequence data, 2-48
- finally statement, 3-42
- `find()` method, strings, 1-61
- First class objects, 2-65
- Float type, 1-46, 1-51
- `float()` function, 1-54
- Floating point numbers, 1-51
- Floating point, accuracy, 1-52
- Floor division operator, 1-50
- `__floordiv__()` method, 5-28
- `for` loop, and tuples, 2-44
- `for` loop, keeping a loop counter, 2-42
- `for` statement, 2-36
- `for` statement, and files, 1-77

- for statement, and generators, 8-10
- for statement, and iteration, 8-2
- for statement, internal operation of, 8-4
- for statement, iteration variable, 2-37
- Format codes, string formatting, 2-27
- format() function, 2-28
- format() method of strings, 9-8
- Formatted output, 2-25
- from module import \*, 4-10
- from statement, 4-9
- from `__future__` import, 1-50
- Function, and generators, 8-10
- Functions, 3-7, 3-8
- Functions, accepting any combination of arguments, 10-5
- Functions, argument passing, 3-11, 3-12
- Functions, benefits of using, 3-7
- Functions, Bottom-up style, 3-10
- Functions, calling with positional arguments, 3-15
- Functions, closures, 10-19
- Functions, conformance checking, 3-28
- functions, decorators, 10-25
- Functions, default arguments, 3-14
- Functions, defining, 1-83
- Functions, definition order, 3-9
- Functions, design of argument names, 3-17
- Functions, global variables, 3-22, 3-23
- Functions, keyword arguments, 3-16
- Functions, lazy evaluation, 10-22
- Functions, local variables, 3-21
- Functions, multiple return values, 3-19
- Functions, returning as a result, 10-17
- Functions, side effects, 3-25, 3-26
- Functions, tuple and dictionary expansion, 10-6
- Functions, variable number of arguments, 10-3, 10-4

## G

- Garbage collection, 1-32
- Generating text, 9-3
- Generator, 8-10, 8-11
- Generator expression, 8-22, 8-23, 8-24
- Generator tricks presentation, 8-30
- Generator, and code reuse, 8-27
- Generator, and StopIteration exception, 8-13
- Generator, producer-consumer problems, 8-15
- Generator, use of, 8-26
- Generators, processing pipelines, 8-16
- getatime() function,, 4-34

- getattr() function, 5-35
- `__getitem__()` method, 5-29
- getmtime() function, `os.path` module, 4-34
- getrefcount() function, `sys` module, 2-60
- getsize() function, `os.path` module, 4-34
- glob module, 4-30
- global statement, 3-24
- Global variables, 3-20, 4-5, 4-6
- Global variables, accessing in functions, 3-22
- Global variables, modifying inside a function, 3-23
- Guido van Rossum, 1-2

## H

- hasattr() function, 5-35
- Hash table, 2-12
- Haskell, list comprehension, 2-51
- has\_key() method, of dictionaries, 2-21
- help() command, 1-16

## I

- Identifiers, 1-30
- IDLE, 1-9, 1-10
- IDLE, creating new program, 1-19, 1-20
- IDLE, on Mac or Unix, 1-11
- IDLE, running programs, 1-22
- IDLE, saving programs, 1-21
- IEEE 754, 1-51
- if statement, 1-37
- Ignoring an exception, 3-37
- Immutable objects, 1-62
- import statement, 1-84, 4-3
- import statement, from modifier, 4-9
- import statement, importing all symbols, 4-10
- import statement, repeated, 4-18
- import statement, search path, 4-19, 4-20
- import, as modifier, 4-8
- in operator, dictionary, 2-21
- in operator, lists, 1-69
- in operator, strings, 1-59
- Indentation, 1-33, 1-34
- Indentation style, 1-35, 1-36
- index() method, strings, 1-61
- Indexing of lists, 1-68
- Infinite data streams, 8-28
- Inheritance, 5-11, 5-14
- Inheritance example, 5-13



- Inheritance, and isa relationship, 5-17
- Inheritance, and object base, 5-18
- Inheritance, and polymorphism, 5-22
- Inheritance, and `__init__()` method, 5-20
- Inheritance, implementation of, 6-17, 6-19
- Inheritance, multiple, 5-23, 6-20, 6-22, 6-23, 6-24, 6-25
- Inheritance, multiple inheritance, 6-21
- Inheritance, organization of objects, 5-15, 5-16
- Inheritance, redefining methods, 5-19, 5-21
- Inheritance, uses of, 5-12
- Initialization of objects, 5-6
- `__init__()` method in classes, 5-6
- `__init__()` method, and inheritance, 5-20
- `__init__.py` files, 10-43
- `insert()` method, of lists, 1-67
- Instance data, 5-7
- Instances, and `__class__` attribute, 6-9
- Instances, creating new, 5-5
- Instances, modifying after creation, 6-13
- Instances, representation of, 6-6, 6-7
- `int()` function, 1-54
- Integer division, 1-50
- Integer type, 1-46
- Integer type, operations, 1-49
- Integer type, precision of, 1-48
- Interactive mode, 1-12, 1-13, 1-14
- Interactive mode, last result, 1-15
- Interpreter prompts, 1-14
- `__invert__()` method, 5-28
- is operator, 2-60
- Isa relationship and inheritance, 5-17
- `isalpha()` method, strings, 1-61
- `isdigit()` method, strings, 1-61
- `isdir()` function, `os.path` module, 4-31, 4-32, 4-33
- `isfile()` function, `os.path` module, 4-31, 4-32, 4-33
- `isinstance()` function, 2-64
- `islower()` method, strings, 1-61
- Item access methods, 5-29
- Iterating over a sequence, 2-36
- Iteration, 8-2
- Iteration protocol, 8-4
- Iteration variable, for loop, 2-37
- iteration, over lists, 1-72
- Iteration, supporting in classes, 8-6
- Iteration, user defined, 8-9
- `itertools` module, 8-7, 8-29
- `__iter__()` method, 8-4

## J

- `join()` function, `os.path` module, 4-31
- `join()` method, of strings, 9-5
- `join()` method, strings, 1-61
- `json` module, 4-59
- JSON parsing, 4-59

## K

- key argument to `sort()`, 10-10
- key function, sorting, 10-10
- `KeyboardInterrupt` exception, 3-44
- Keys, dictionary, 2-12
- Keyword arguments, 3-16
- Keywords, 1-31

## L

- lambda expression, 10-12
- lambda expression, uses of, 10-13
- Lazy evaluation, 8-28, 10-22
- `len()` function, 2-31, 5-29
- `len()` function, lists, 1-69
- `len()` function, strings, 1-59
- `__len__()` method, 5-29
- Library modules, 1-84
- Line continuation, 1-43
- List, 2-31
- List comprehension, 2-47, 2-48, 2-49
- List comprehension uses, 2-50
- list comprehension, versus `map()`, 10-14
- List comprehensions and `awk`, 2-52
- List concatenation, 1-67
- List replication, 1-69
- List type, 1-67
- `list()` function, 2-61
- List, Looping over items, 2-36
- List, math operations, 1-74
- List, sorting, 1-73
- List, use as a container, 2-17, 2-18
- `listdir()` function, `os` module, 4-30
- Lists, changing elements, 1-68
- Lists, indexing, 1-68
- Lists, iterating over, 1-72
- Lists, removing items, 1-70, 1-71
- Lists, searching, 1-69

- lists, sorting, 10-8
- Local variables, 3-20
- Local variables in functions, 3-21
- log() function, math module, 1-53
- logging module, 7-17
- Long type, 1-46
- long() function, 1-54
- Looping over integers, 2-40
- Looping over items in a sequence, 2-36
- Looping over multiple sequences, 2-45
- lower() method, strings, 1-60, 1-61
- \_\_lshift\_\_() method, 5-28

## M

- Main program, 4-14, 4-15, 4-16, 4-17
- main() function, unittest module, 7-13
- \_\_main\_\_, 4-14, 4-15, 4-16, 4-17
- \_\_main\_\_ module, 7-8
- map() function, 10-14
- math module, 1-53, 1-84
- Math operators, 5-28
- Math operators, floating point, 1-53
- Math operators, integer, 1-49
- matplotlib, 4-126
- matplotlib, plotting example, 4-127, 4-128
- matrices, numpy extension, 4-113
- max() function, 2-35
- Memory efficiency, and generators, 8-28
- Method invocation, 5-32
- Method, definition of, 5-2, 5-3
- Methods, calling in base class, 6-26
- Methods, calling other methods in the same class, 5-9
- Methods, in classes, 5-8
- min() function, 2-35
- mock, 7-15
- Modules, 4-3
- modules variable, sys module, 4-18
- Modules, execution of, 4-7
- Modules, loading of, 4-18
- Modules, namespaces, 4-4
- Modules, search path, 4-19, 4-20
- Modules, self-testing with doctest, 7-8
- \_\_mod\_\_() method, 5-28
- move() function, shutil module, 4-36
- \_\_mro\_\_ attribute, of classes, 6-20, 6-22, 6-23, 6-24, 6-25
- Multiple inheritance, 5-23, 6-20, 6-21, 6-22, 6-23, 6-24, 6-25

- \_\_mul\_\_() method, 5-28

## N

- Namespaces, 4-4
- \_\_name\_\_ attribute, of modules, 4-14, 4-15, 4-16, 4-17
- Naming conventions, Python's reliance upon, 6-30
- Negative indices, lists, 1-68
- \_\_neg\_\_() method, 5-28
- Nested functions, 10-17
- next() method, and iteration, 8-4
- next() method, of generators, 8-12
- no-op statement, 1-42
- None type, 2-4
- None type, returned by functions, 3-18
- nose, 7-15
- not operator, 1-38, 1-39
- Notable third party packages, 4-83
- Null value, 2-4
- Numeric conversion, 1-54
- Numeric datatypes, 1-46
- numpy, 4-95, 4-116, 4-117, 4-118, 4-119, 4-120, 4-121, 4-122, 4-123, 4-124
- numpy, array access, 4-100
- numpy, array math, 4-107
- numpy, arrays, 4-96
- numpy, conditionals, 4-111
- numpy, matrices, 4-113
- numpy, universal functions, 4-109

## O

- object base class, 5-18
- object identity, 2-60
- Object oriented programming, 5-2, 5-3
- Object oriented programming, and encapsulation, 6-28
- Objects, attribute binding, 6-15, 6-18
- Objects, attributes of, 5-7
- Objects, calling methods in base class, 6-26
- Objects, creating containers, 5-29
- Objects, creating new instances, 5-5
- Objects, creating private attributes, 6-31, 6-32, 6-33
- Objects, defining new, 5-4
- Objects, first class behavior, 2-65
- objects, getting the reference count, 2-60
- Objects, inheritance, 5-11
- Objects, invoking methods, 5-5
- Objects, making a deep copy of, 2-62

- Objects, method invocation, 5-32
- Objects, modifying attributes of instances, 6-12
- Objects, modifying instances, 6-13
- Objects, multiple inheritance, 6-21
- Objects, reading attributes, 6-14
- Objects, representation of, 6-10
- Objects, representation of instances, 6-6, 6-7
- Objects, representation with dictionary, 6-5
- Objects, saving with pickle, 4-72
- Objects, single inheritance, 6-19
- Objects, special methods, 5-25
- Objects, type checking, 2-64
- Objects, type of, 2-63
- Old-style classes, 5-18
- Online help, 1-16
- open() function, 1-76
- Optimized mode, 7-28
- Optimized mode (-O), 7-27
- Optional features, defining function with, 3-16
- Optional function arguments, 3-14
- or operator, 1-38, 1-39
- organizing code into a package, 10-41
- \_\_or\_\_() method, 5-28
- os module, 4-28
- os.path module, 4-31, 4-34
- os.walk() function, 4-35
- Output, print statement, 1-40

## P

- packages, 10-41
- Packing values into a tuple, 2-9
- Parallel iteration, 2-45
- pass statement, 1-42
- path variable, sys module, 4-19, 4-20
- pdb module, 7-30
- pdb module, commands, 7-32
- Performance statistics, profile module, 7-35
- Perl, difference in string handling, 1-82
- Perl, string interpolation and Python, 9-7
- pickle module, 4-72
- Pipelines, and generators, 8-26
- Pipelines, creating with generators, 8-16
- plotting, matplotlib example, 4-127, 4-128
- Polymorphism, and inheritance, 5-22
- pop() method, of deques, 4-73
- popleft() method, of deques, 4-73
- Positional function arguments, 3-15
- Post-assertion, 7-26

- pow() function, 1-49, 1-53
- Powers of numbers, 1-49
- \_\_pow\_\_() method, 5-28
- Pre-assertion, 7-26
- Primitive datatypes, 2-3
- print statement, 1-40, 4-24, 4-25, 5-26, 5-27
- print statement, and files, 1-76
- print statement, and str(), 1-63
- print statement, trailing comma, 1-40
- print, formatted output, 2-26, 2-29
- Private attributes, 6-31, 6-32, 6-33
- Producer-consumer problem, with generators, 8-15
- profile module, 7-35
- Profiling, 7-35
- Program exit, 3-44
- Program structure, definition order, 3-6
- Propagation of exceptions, 3-31, 3-32, 3-33
- Properties, and encapsulation, 6-36
- @property decorator, 6-36, 10-34
- property() function, 6-43
- py files, 1-18
- Python Documentation, 1-16
- Python eggs packages, 4-88
- Python interpreter, 1-12
- Python interpreter, keeping alive after execution, 7-29
- Python interpreter, optimized mode, 7-27, 7-28
- Python package index, 4-82
- python versions, 1-4
- Python, reason created, 1-5
- Python, running on command line, 1-23
- Python, source files, 1-18
- Python, starting on Mac, 1-11
- Python, starting on Unix, 1-11
- Python, starting on Windows, 1-10
- Python, starting the interpreter, 1-8
- Python, statement execution, 1-28
- Python, uses of, 1-6, 1-7
- Python, year created, 1-2
- python.org website, 1-3

## R

- raise Statement, 1-87
- raise statement, 3-30
- Raising exceptions, 1-87
- range() function, 2-41
- range() vs. xrange(), 2-41
- Raw strings, 1-56, 1-64
- raw\_input() function, 1-41

- re module, 4-42
- Read-eval loop, 1-13
- Reading attributes on objects, 6-14
- Reading from keyboard, 1-41
- readline() method, files, 1-76
- Redefining methods with inheritance, 5-19, 5-21
- Redefining output file, 4-24, 4-25
- Redirecting print to a file, 1-76
- referenc count, obtaining on an object, 2-60
- Reference counting, 2-61
- regular expressions, 4-42
- Relational operators, 1-38
- remove() method, lists, 1-70, 1-71
- Repeated imports, 4-18
- replace() method, strings, 1-60, 1-61
- Replacing text, 1-60
- Replication of sequences, 2-32
- repr() function, 5-26, 5-27
- Representation of strings, 1-57
- \_\_repr\_\_() method, 5-26, 5-27
- Reserved names, 1-31
- return statement, 3-18
- return statement, multiple values, 3-19
- rfind() method, strings, 1-61
- rindex() method, strings, 1-61
- rmtree() function, shutil module, 4-36
- Rounding errors, floating point, 1-52
- \_\_rshift\_\_() method, 5-28
- Ruby, string interpolation and Python, 9-7
- run() function, pdb module, 7-33, 7-34
- runcall() function, pdb module, 7-33, 7-34
- runeval() function, pdb module, 7-33, 7-34
- Running Python, 1-8
- Runtime error vs. compile-time error, 3-29

## S

- safe\_substitute() method, of Template objects, 9-9
- Sample Python program, 1-25
- Scope of iteration variable in loops, 2-37
- Scripting, 3-3
- Scripting language, 1-2
- Scripting, defined, 3-4
- Scripting, problem with, 3-5
- self parameter of methods, 5-7, 5-8
- Sequence, 2-31
- Sequence, concatenation, 2-32
- Sequence, extended slicing, 2-34
- Sequence, indexing, 2-31
- Sequence, length, 2-31
- Sequence, looping over items, 2-36
- Sequence, replication, 2-32
- Sequence, slicing, 2-33
- Sequence, string, 1-57
- Set theory, list comprehension, 2-51
- Set type, 2-22, 2-23
- set() function, 2-22, 2-23
- setattr() function, 5-35
- \_\_setitem\_\_() method, 5-29
- setup.py file, third party modules, 4-87
- setuptools module, 4-88
- set\_trace() function, pdb module, 7-31
- Shallow copy, 2-61
- Shell operations, 4-36
- shutil module, 4-36
- Side effects, 3-25, 3-26
- sin() function, math module, 1-53
- Single-quoted string, 1-55
- Slicing operator, 2-33
- \_\_slots\_\_ attribute of classes, 6-44
- sort() method, of lists, 1-73
- sort() method, of Lists, 10-10
- Sorting lists, 1-73
- Source files, 1-18
- Source files, and modules, 4-3
- Special methods, 5-25
- split() method, strings, 1-61, 1-66
- Splitting text, 1-66
- sqrt() function, math module, 1-53
- Standard I/O streams, 4-24, 4-25
- Standard library, 4-22
- startswith() method, strings, 1-61
- Statements, 1-28
- static methods, 10-35, 10-36
- @staticmethod decorator, 10-34, 10-35, 10-36
- stderr variable, sys module, 4-24, 4-25
- stdin variable, sys module, 4-24, 4-25
- stdout variable, sys module, 4-24, 4-25
- StopIteration exception, 8-5
- StopIteration exception, and generators, 8-13
- str() function, 1-63, 5-26, 5-27
- String concatenation, performance of, 9-4
- String format codes, 2-27
- String formatting, 2-26, 2-29
- String interpolation, 9-7
- String joining, 9-5
- String joining vs. concatenation, 9-6
- String templates, 9-9

- String type, 2-31
- strings, case insensitive sorting, 10-9
- Strings, concatenation, 1-58
- Strings, conversion to, 1-63
- Strings, conversion to numbers, 1-54, 1-82
- Strings, escape codes, 1-55, 1-56, 1-64
- Strings, format() function, 2-28
- Strings, immutability, 1-62
- Strings, indexing, 1-58
- Strings, length of, 1-59
- Strings, literals, 1-55
- Strings, methods, 1-60, 1-61
- Strings, raw strings, 1-56, 1-64
- Strings, replication, 1-59
- Strings, representation, 1-57
- Strings, searching for substring, 1-59
- Strings, slicing, 1-58
- Strings, splitting, 1-66
- Strings, triple-quoted, 1-55
- strip() method, strings, 1-60, 1-61
- Stripping characters, 1-60
- Subclass, 5-11
- substitute() method, of Template objects, 9-9
- \_\_sub\_\_() method, 5-28
- sum() function, 2-35
- super() function, 6-26, 6-27
- Superclass, 5-11
- sys module, 4-23
- system() function, os module, 4-28
- SystemExit exception, 3-44

## T

- tan() function, math module, 1-53
- Template strings, 9-9
- TestCase class, of unittest module, 7-10
- Testing, 7-3
- Testing files, 4-32, 4-33
- testmod() function, doctest module, 7-6
- Text replacement, 1-60
- Text strings, 1-57
- Third party modules, 4-82, 4-84
- Third party modules, and C/C++ code, 4-89
- Third party modules, eggs format, 4-88
- Third party modules, native installer, 4-85
- Third party modules, setup.py file, 4-87
- Third party modules, system installer, 4-86
- time module, 4-34, 4-37
- Tracebacks, 1-87

- Triple-quoted string, 1-55
- True value, 1-47
- Truncation, integer division, 1-50
- Truth values, 1-38, 1-39
- try statement, 1-86, 3-30
- Tuple, 2-6, 2-31
- Tuple, immutability of, 2-8
- Tuple, packing values, 2-9
- Tuple, unpacking in for-loop, 2-44
- Tuple, unpacking values, 2-10
- Tuple, use of, 2-7
- Tuple, using as function arguments, 10-6
- Type checking, 2-64
- Type conversion, 1-82
- Type of objects, 2-63
- type() function, 2-63, 2-64
- Types, 1-30
- Types, numeric, 1-46
- Types, primitive, 2-3

## U

- Uniform access principle, 6-41
- Unit testing, 7-9
- unittest module, 7-9, 7-10
- unittest module, example of, 7-11
- unittest module, running tests, 7-13
- universal functions, numpy, 4-109
- Unpacking values from tuple, 2-10
- upper() method, strings, 1-60, 1-61
- urllib module, 1-84
- urlopen() function, urllib module, 1-84
- User input, 1-41
- User-defined exceptions, 5-31

## V

- Value of exceptions, 3-35
- Variable assignment, 2-56, 2-58
- Variable assignment, assignment of globals in function, 3-24
- Variable assignment, global vs. local, 3-20
- Variable number of function arguments, 10-3, 10-4
- Variables, and modules, 4-5, 4-6
- Variables, lifetime of, 1-32
- Variables, names of, 1-30
- Variables, scope in functions, 3-21, 3-22
- Variables, type of, 1-30

## **W**

walking over a directory tree, 4-35  
where() function, numpy, 4-111  
while statement, 1-33  
Windows, starting Python, 1-10  
with statement, 1-80, 3-43  
write() method, files, 1-76

## **X**

XML parsing, 4-52  
xml.etree.ElementTree module, 4-54  
\_\_xor\_\_() method, 5-28  
xrange() function, 2-40  
xrange() vs. range(), 2-41

## **Z**

zeros() function, numpy, 4-96  
zip() function, 2-45