Python in Action

Presented at USENIX LISA Conference
November 16, 2007

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

(Part I - Introducing Python)

Course Overview

- Python Programming by example in two acts
  - Part I : The Python Language
  - Part II : Python Systems Programming
- "In Action" means doing useful things.

Prerequisites

- I'm going to assume that...
  - you have written programs
  - you know about basic data structures
  - you know what a function is
  - you know about basic system concepts (files, I/O, processes, threads, network, etc.)
- I do not assume that you know Python

Target Audience

- This tutorial is aimed at programmers who want to get some idea of what Python is all about.
- I assume that you're interested in solving practical problems.
- Tool building

My Background

- C/assembler programming
- Started using Python in 1996 as a control language for physics software running on supercomputers at Los Alamos.
- Author: "Python Essential Reference"
- Developer of several open-source packages
- Currently working on parsing/compiler writing tools for Python.

What is Python?

- An interpreted, dynamically typed programming language.
- In other words: A language that's similar to Perl, Ruby, Tcl, and other so-called "scripting languages."
- Created by Guido van Rossum around 1990.
- Named in honor of Monty Python

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

Important Influences

- C (syntax, operators, etc.)
- ABC (syntax, core data types, simplicity)
- Unix ("Do one thing well")
- Shell programming (but not the syntax)
- Lisp, Haskell, and Smalltalk (later features)

Some Uses of Python

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

More than "Scripting"

- Although Python is often used for "scripting", it is a general purpose programming language
- Major applications are written in Python
- Large companies you have heard of are using hundreds of thousands of lines of Python.

Our Focus : Systems

- In this tutorial we will cover a slice of Python
  - Language introduction
  - Data processing/parsing
  - Files and I/O
  - Systems programming

Notable Omissions

- Object-oriented programming. Python fully supports objects, but covering this would require an entire class. Besides, it's not needed to write useful programs.
- Web frameworks. There are a variety of frameworks for building web sites and Internet programming in Python. This too, would require a dedicated class.

Getting Started

Where to get Python?

http://www.python.org

- Site for downloads, community links, etc.
- Current version: Python-2.5.1
- Supported on virtually all platforms

Support Files

- Program files, examples, and datafiles for this tutorial are available here:

  http://www.dabeaz.com/action

- Please go there and follow along

Running Python (Unix)

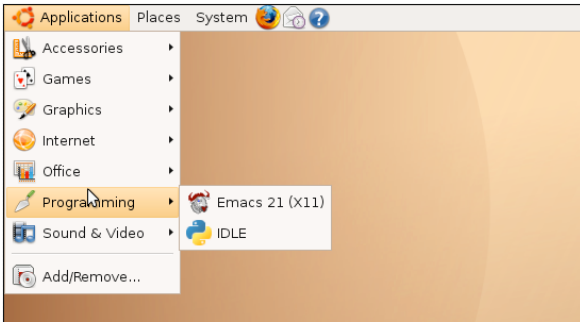
- Command line

```
shell % python
Python 2.5.1 (r251:54869, Apr 18 2007, 22:08:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)] on darwin
Type "help", "copyright", "credits" or "license"
>>>
```

- Integrated Development Environment (IDLE)

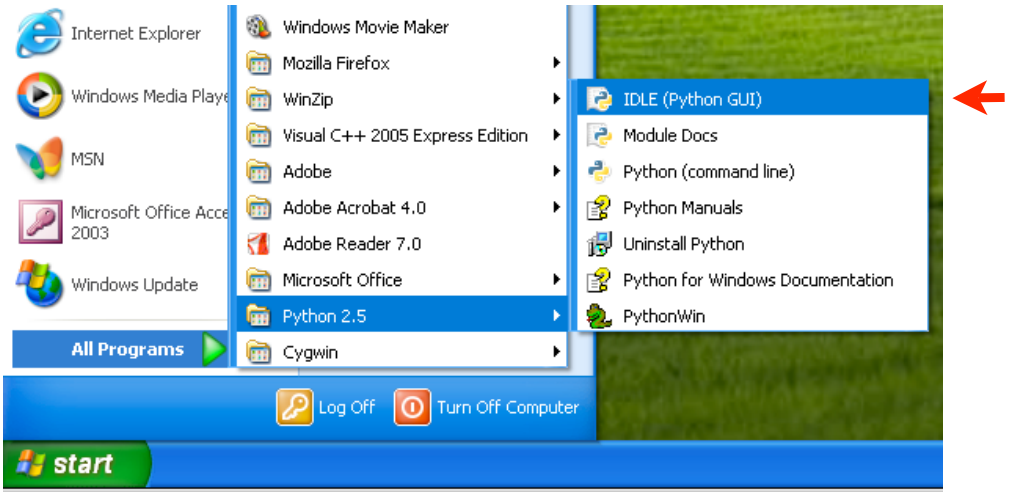
```
shell % idle
```

or



Running Python (win)

- Start Menu (IDLE or PythonWin)



Python Interpreter

- All programs execute in an interpreter
- If you give it a filename, it interprets the statements in that file in order
- Otherwise, you get an "interactive" mode where you can experiment
- No separate compilation step

Interactive Mode

- 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
- This is one of the most useful features

Creating Programs

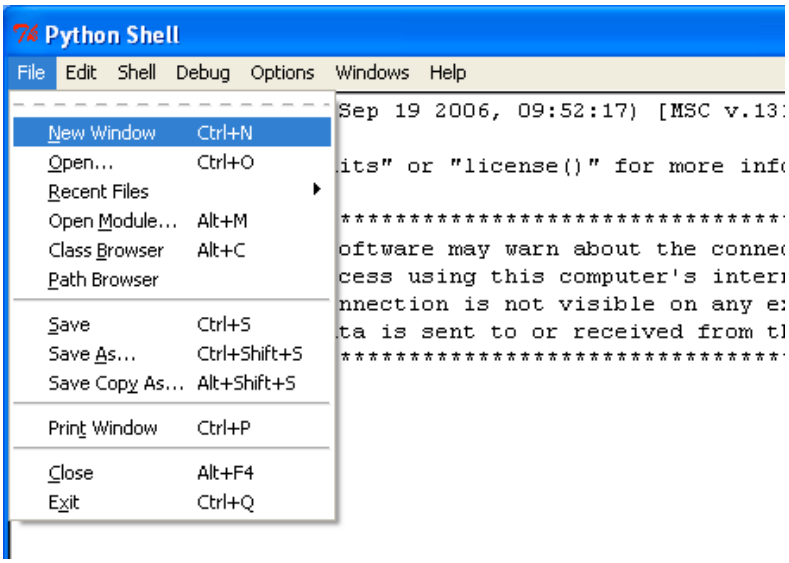
- Programs are put in .py files

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

- Source files are simple text files
- Create with your favorite editor (e.g., emacs)
- Note: There may be special editing modes
- There are many IDEs (too many to list)

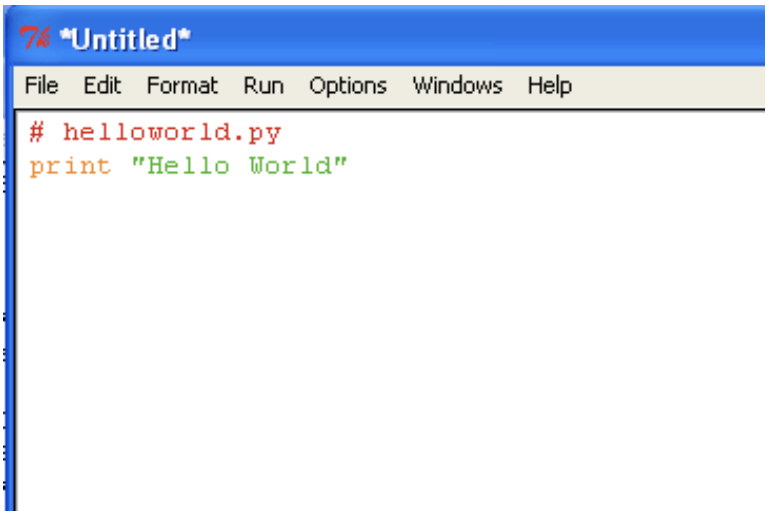
Creating Programs

- Creating a new program in IDLE



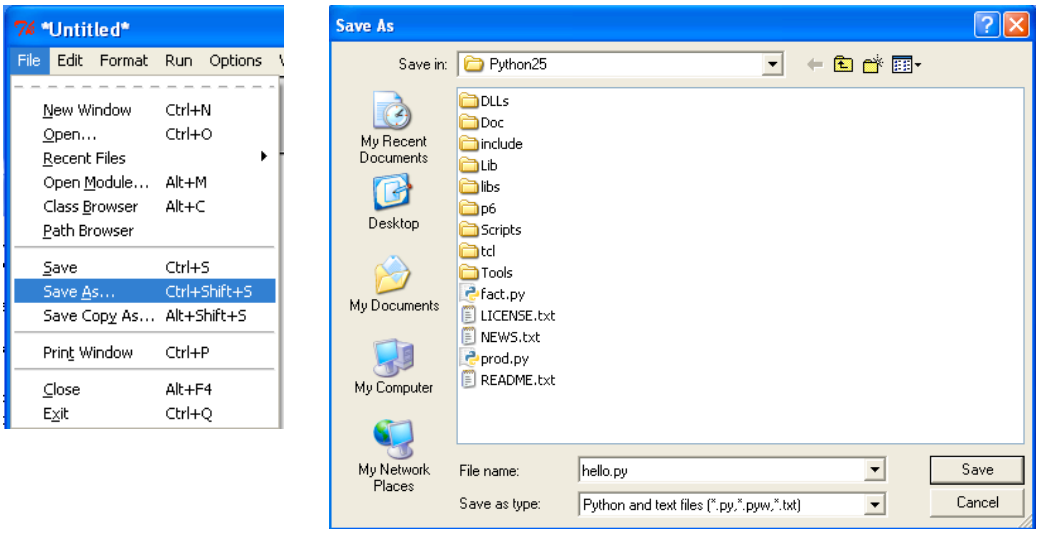
Creating Programs

- Editing a new program in IDLE



Creating Programs

- Saving a new Program in IDLE



Running Programs

- In production environments, Python may be run from command line or a script
- Command line (Unix)

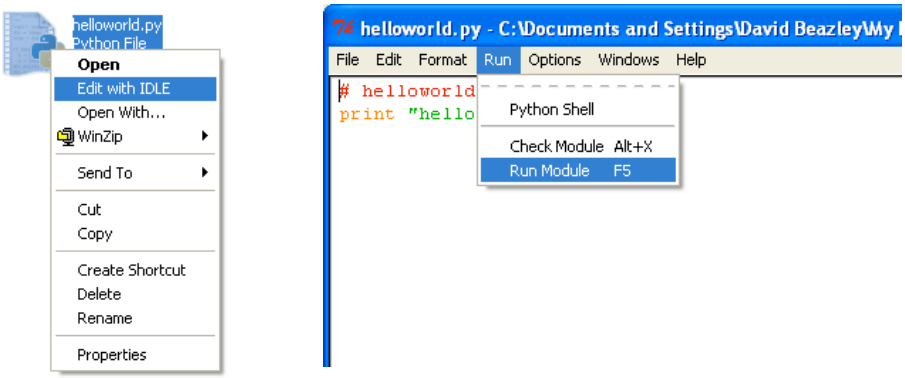
```
shell % python helloworld.py
hello world
shell %
```

- Command shell (Windows)

```
C:\Somewhere>c:\python25\python helloworld.py
hello world
C:\Somewhere>
```

Running Programs (IDLE)

- Select "Run Module" (F5)



- Will see output in IDLE shell window

A Sample Program

- Dave's Mortgage

  Dave has taken out a $500,000 mortgage from Guido's Mortgage, Stock, and Viagra trading corporation. He got an unbelievable rate of 4% and a monthly payment of only $499. However, Guido, being kind of soft-spoken, didn't tell Dave that after 2 years, the rate changes to 9% and the monthly payment becomes $3999.

- Question: How much does Dave pay and how many months does it take?

mortgage.py

```
# mortgage.py

principle  = 500000        # Initial principle
payment    = 499           # Monthly payment
rate       = 0.04          # The interest rate
total_paid = 0             # Total amount paid
months     = 0             # Number of months

while principle > 0:
      principle = principle*(1+rate/12) - payment
      total_paid += payment
      months     += 1
      if months == 24:
          rate    = 0.09
          payment = 3999

print "Total paid", total_paid
print "Months", months
```


Python 101: Statements

```
# mortgage.py

principle  = 500000
payment    = 499
rate       = 0.04
total_paid = 0
months     = 0             # Number of months

while principle > 0:
      principle = principle*(1+rate/12) - payment
      total_paid += payment
      months     += 1
      if months == 24:
          rate    = 0.09
          payment = 3999

print "Total paid", total_paid
print "Months", months
```

Each statement appears on its own line

No semicolons

Python 101: Comments

```
# mortgage.py

principle  = 500000
payment    = 499
rate       = 0.04
total_paid = 0
months     = 0

                                    payment

          rate    = 0.09
          payment = 3999

print "Total paid", total_paid
print "Months", months
```

```
# Initial principle
# Monthly payment
# The interest rate
# Total amount paid
# Number of months
```

starts a comment which extends to the end of the line

Python 101: Variables

```
# mortgage.py

principle  = 500000
payment    = 499
rate       = 0.04
total_paid = 0
months     = 0

while principle > 0:
      principle = princ
      total_paid += pay
      months     += 1
      if months == 24:
          rate    = 0.0
          payment = 399

print "Total paid", tot
print "Months", months
```

Variables are declared by assigning a name to a value.

- Same name rules as C ([a-zA-Z_][a-zA-Z0-9_]*)
- You do not declare types like int, float, string, etc.
- Type depends on value

Python 101: Keywords

```
# mortgage.py

principle
payment
rate
total_paid
months

while principle > 0
      principle = p
      total_paid +=
      months     +=
      if months ==
          rate    =
          payment =

print "Total paid",
print "Months", mon
```

Python has a small set of keywords and statements

Keywords are C-like

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

Python 101: Looping

while executes a loop as long as a condition is True

```
while expression:
    statements
    ...
```

```
principle
payment
erest rate
mount paid
of months
```

```
while principle > 0:
      principle = principle*(1+rate/12) - payment
      total_paid += payment
      months     += 1
      if months == 24:
          rate    = 0.09
          payment = 3999

print "T
print "M
```

loop body denoted by indentation

Python 101: Conditionals

if-elif-else checks a condition

```
if expression:
   statements
   ...
elif expression:
   statements
   ...
else:
   statements
   ...
```

```
                                    al principle
                                    nly payment
                                    nterest rate
                                    l amount paid
                                    er of months

                                    te/12) - payment
        months      += 1
        if months == 24:
            rate    = 0.09
            payment = 3999

print "Total
print "Months
```

body of conditional denoted by indentation

Python 101: Indentation

```
# mortgage.py

principle  = 500000        # Initial principle
payment    = 499
rate       = 0.04
total_paid = 0
months     = 0

while principle > 0:
      principle = principle*(1+rate/12) - payment
      total_paid += payment
      months     += 1
      if months == 24:
          rate    = 0.09
          payment = 3999

print "Total paid", total_paid
print "Months", months
```

: indicates that an indented block will follow

Python 101: Indentation

```
# mortgage.py

principle  = 500000        # Initial principle
payment    = 499           # Monthly payment
rate       = 0.04          # The interest rate
total_paid = 0             # Total amount paid
months     = 0             # Number of months

while principle > 0:
      principle = principle*(1+rate/12) - payment
      total_paid += payment
      months     += 1
      if months == 24:
          rate    = 0.09
          payment = 3999
```

Python only cares about consistent indentation in the same block

Python 101: Primitive Types

```
# mortgage.py

principle  = 500000
payment    = 499
rate       = 0.04
total_paid = 0
months     = 0

while principle > 0:
      principle = principle*(1+rate/12) - payment
      total_paid += payment
      months     += 1
      if months == 24:
          rate    = 0.09
          payment = 3999

print "Total paid", total_paid
print "Months", months
```

Numbers:
- Integer
- Floating point

Strings

Python 101: Expressions

```
# mortgage.py
```

Python uses conventional syntax for operators and expressions

```
ial principle
hly payment
interest rate
l amount paid
er of months

while principle > 0:
      principle = principle*(1+rate/12) - payment
      total_paid += payment
      months     += 1
      if months == 24:
          rate    = 0.09
          payment = 3999

print "Total paid", total_p
print "Months", months
```

Basic Operators

```
+ - * / // % ** << >> | & ^
< > <= >= == != and or not
```


Python 101: Output

```
# mortgage.py

principle  = 500000        # Initial principle
payment    = 499           # Monthly payment
rate       = 0.04          # The interest rate
total_paid = 0             # Total amount paid
months     = 0             # Number of months
```

print writes to standard output
- Items are separated by spaces
- Includes a terminating newline
- Works with any Python object

```
print "Total paid", total_paid
print "Months", months
```

Running the Program

- Command line

```
shell % python mortgage.py
Total paid 2623323
Months 677
shell %
```

- Keeping the interpreter alive (-i option or IDLE)

```
shell % python -i mortgage.py
Total paid 2623323
Months 677
>>> months/12
56
>>>
```

- In this latter mode, you can inspect variables and continue to type statements.

Interlude

- If you know another language, you already know a lot of Python
- Python uses standard conventions for statement names, variable names, numbers, strings, operators, etc.
- There is a standard set of primitive types such as integers, floats, and strings that look the same as in other languages.
- Indentation is most obvious "new" feature

Getting Help

- Online help is often available
- help() command (interactive mode)

```
IDLE 1.2
>>> 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 integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.

>>>
```

- Documentation at http://www.python.org

dir() function

- dir() returns list of symbols

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__',
'__name__', '__stderr__', '__stdin__', '__stdout__',
'_current_frames', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing',
'callstats', 'copyright', 'displayhook', 'exc_clear',
'exc_info', 'exc_type', 'excepthook', 'exec_prefix',
'executable', 'exit', 'getcheckinterval',
 ...
'version_info', 'warnoptions']
```

- Useful for exploring, inspecting objects, etc.

More on Relations

- 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"
```

- Don't use &&, ||, and ! as in C

```
&&  ⟶  and
||  ⟶  or
!   ⟶  not
```

- Relations do not require surrounding ( )

Line Continuation

- Line continuation for long statements (\)

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

- Line continuation is not needed for any code inside (), [], or { }

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

More on Numbers

- Numeric Datatypes

```
a = True          # A boolean (True or False)
b = 42            # An integer (32-bit signed)
c = 81237742123L  # A long integer (arbitrary precision)
d = 3.14159       # Floating point (double precision)
```

- Integer operations that overflow become longs

```
>>> 3 ** 73
675851986348175232355204436243179 23L
>>> a = 72883988882883812
>>> a
72883988882883812L
>>>
```

- Integer division truncates (for now)

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


More on Strings

- String literals use several quoting styles

```
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 sequences work (e.g., '\n')
- Triple quotes capture all literal text enclosed

Basic String Manipulation

- Length of a string

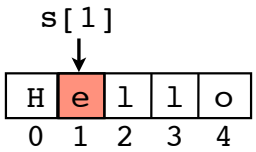
```
n = len(s)        # Number of characters in s
```

- String concatenation

```
s = "Hello"
t = "World"
a = s + t         # a = "HelloWorld"
```

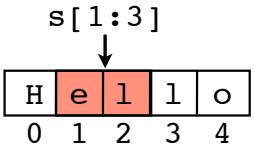
- Strings as arrays : s[*n*]

```
s = "Hello"
s[1]  ⟶ 'e'
s[-1] ⟶ 'o'
```



- Slices : s[*start:end*]

```
s[1:3] ⟶ "el"
s[:4]  ⟶ "Hell"
s[-4:] ⟶ "ello"
```



Type Conversion

- Converting between data types

```
a = int(x)        # Convert x to an integer
b = long(x)       # Convert x to a long
c = float(x)      # Convert x to a float
d = str(x)        # Convert x to a string
```

- Examples:

```
>>> int(3.14)
3
>>> str(3.14)
'3.14'
>>> int("0xff")
255
>>>
```

Programming Problem

- Dave's stock scheme

  After watching 87 straight hours of "Guido's Insane Money" on his Tivo, Dave hatched a get rich scheme and purchased a bunch of stocks.

  He can no longer remember the evil scheme, but he still has the list of stocks in a file "portfolio.dat".

- Write a program that reads this file, prints a report, and computes how much Dave spent during his late night stock "binge."

The Input File

- Input file: portfolio.dat

```
IBM      50      91.10
MSFT    200      51.23
GOOG    100     490.10
AAPL     50     118.22
YHOO     75      28.34
SCOX    500       2.14
RHT      60      23.45
```

- The data: Name, Shares, Price per Share

portfolio.py

```
# portfolio.py

total = 0.0
f     = open("portfolio.dat","r")

for line in f:
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float(fields[2])
    total += shares*price
    print  "%-10s %8d %10.2f" % (name,shares,price)

f.close()
print "Total", total
```


Python File I/O

```
# portfolio.py

total = 0.0
f     = open("portfolio.dat","r")

for line in
    fields =
    name   =
    shares =
    price  =
    total +=
    print  "

f.close()
print "Total", total
```

```
"r"  - Read
"w"  - Write
"a"  - Append
```

Files are modeled after C stdio.

- f = open() - opens a file
- f.close() - closes the file

Data is just a sequence of bytes

Reading from a File

```
# portfolio.py

total = 0.0
f     = open("p

for line in f:
    fields = line.split()
    name   = fields[0]
    shares = in
    price  = fl
    total += sh
    print  "%-1

f.close()
print "Total",
```

Loops over all lines in the file.
Each line is returned as a string.

Alternative reading methods:

- f.read([nbytes])
- f.readline()
- f.readlines()

String Processing

```
# portfolio.

total = 0.0
f     = open

for line in f:
    fields = line.split()
    name   = fields[0]
    shares = int(
    price  = floa
    total += shar
    print  "%-10s

f.close()
print "Total", to
```

Strings have various "methods."
split() splits a string into a list of strings

```
line = 'IBM     50     91.10\n'

          fields = line.split()

fields = ['IBM', '50', '91.10']
```

Lists

```
# portfolio.py

total = 0.0
f     = open("por

for line in f:
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float(fields[2])
    total += shares*price
    print  "%-10s %8d

f.close()
print "Total", total
```

A 'list' is an ordered sequence of objects. It's like an array.

```
fields = ['IBM', '50', '91.10']
```


Types and Operators

```
# portfolio.py

total = 0.0
f     = open("portfo

for line in f:
    fields = line.sp
    name   = fields[
    shares = int(fields[1])
    price  = float(fields[2])
    total += shares*price
    print  "%-10s %8d %10.2f" % (name,shares,price)

f.close()
print "Total",
```

To work with data, it must be converted to an appropriate type (e.g., number, string, etc.)

Operators only work if objects have "compatible" types

String Formatting

```
# portfolio.py

total = 0.0
f     = ope

for line i
    fields
    name
    shares = int(fields[1])
    price  = float(fields[2])
    total += shares*price
    print  "%-10s %8d %10.2f" % (name,shares,price)

f.close()
print "Total
```

% operator when applied to a string, formats it. Similar to the C printf() function.

format string

values

Sample Output

```
shell % python portfolio.py
IBM             50      91.10
MSFT           200      51.23
GOOG           100     490.10
AAPL            50     118.22
YHOO            75      28.34
SCOX           500       2.14
RHT             60      23.45
Total 74324.5
shell %
```

More on Files

- Opening a file

```
f = open("filename","r")     # Reading
g = open("filename","w")     # Writing
h = open("filename","a")     # Appending
```

- Reading

```
f.read([nbytes])        # Read bytes
f.readline()            # Read a line
f.readlines()           # Read all lines into a list
```

- Writing

```
g.write("Hello World\n")   # Write text
print >>g, "Hello World"   # print redirection
```

- Closing

```
f.close()
```


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 delimeter
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
```

More on Lists

- A indexed sequence of arbitrary objects

```
fields = ['IBM','50','91.10']
```

- Can contain mixed types

```
fields = ['IBM',50, 91.10]
```

- Can contain other lists:

```
portfolio = [ ['IBM',50,91.10],
              ['MSFT',200,51.23],
              ['GOOG',100,490.10] ]
```


List Manipulation

- Accessing/changing items : s[*n*], s[*n*] = *val*

```
fields = [ 'IBM', 50, 91.10 ]

name  = fields[0]        # name = 'IBM'
price = fields[2]        # price = 91.10
fields[1] = 75           # fields = ['IBM',75,91.10]
```

- Slicing : s[*start:end*], s[*start:end*] = t

```
vals = [0, 1, 2, 3, 4, 5, 6]
vals[0:4] ------------> [0, 1, 2, 3]
vals[-2:] ------------> [5, 6]
vals[:2]  ------------> [0, 1]

vals[2:4] = ['a','b','c']
# vals = [0, 1, 'a', 'b', 'c', 4, 5, 6 ]
```

List Manipulation

- Length : len(s)

```
fields = [ 'IBM', 50, 91.10 ]
len(fields) ⟶  3
```

- Appending/inserting

```
fields.append('11/16/2007')
fields.insert(0,'Dave')

# fields = ['Dave', 'IBM', 50, 91.10, '11/16/2007']
```

- Deleting an item

```
del fields[0]  # fields = ['IBM',50,91.10,'11/16/2007']
```


Some List Methods

```
s.append(x)          # Append x to end of s
s.extend(t)          # Add items in t to end of s
s.count(x)           # Count occurences of x in s
s.index(x)           # Return index of x in s
s.insert(i,x)        # Insert x at index i
s.pop([i])           # Return element i and remove it
s.remove(x)          # Remove first occurence of x
s.reverse()          # Reverses items in list
s.sort()             # Sort items in s in-place
```

Programming Problem

- Dave's stock portfolio

  Dave still can't remember his evil "get rich quick" scheme, but if it involves a Python program, it will almost certainly involve some data structures.

- Write a program that reads the stocks in 'portfolio.dat' into memory. Alphabetize the stocks and print a report. Calculate the initial value of the portfolio.

The Previous Program

```
# portfolio.py

total = 0.0
f     = open("portfolio.dat","r")

for line in f:
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float(fields[2])
    total += shares*price
    print  "%-10s %8d %10.2f" % (name,shares,price)

f.close()
print "Total", total
```

Simplifying the I/O

```
# portfolio.py

total = 0.0

for line in open("portfolio.dat"):
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float(fields[2])
    total += shares*price
    print  "%-10s %8d %10.2f" % (name,shares,price)

print "Total", total
```

Opens a file, iterates over all lines, and closes at EOF.

Building a Data Structure

```
# portfolio.py

stocks = []

for line in open("portfolio.dat"):
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float(fields[2])
    holding= (name,shares,price)
    stocks.append(holding)

# print "Total", total
```

A list of "stocks"

Create a stock record and append to the stock list

Tuples - Compound Data

```
# portfolio.py

stocks = []

for line in ope
    fields = li
    name   = fi
    shares = int(fields[1])
    price  = float(fields[2])
    holding= (name,shares,price)
    stocks.append(holding)

# print "Total", total
```

A tuple is the most primitive compound data type (a sequence of objects grouped together)

How to write a tuple:

```
t = (x,y,z)
t = x,y,z   # ()'s are optional
t = ()      # An empty tuple
t = (x,)    # A 1-item tuple
```


A List of Tuples

```
# portfolio.py

stocks = []

for line in open("p
    fields = line.s
    name   = fields
    shares = int(fi
    price  = float(
    holding= (name,
    stocks.append(h

# print "Total", to
```

```
stocks = [
    ('IBM', 50, 91.10),
    ('MSFT', 200, 51.23),
    ('GOOG', 100, 490.10),
    ('AAPL', 50, 118.22),
    ('SCOX', 500, 2.14),
    ('RHT', 60, 23.45)
]
```

This works like a 2D array

```
stocks[2] ⟶ ('GOOG',100,490.10)
stocks[2][1] ⟶ 100
```

Sorting a List

```
# portfolio.py

stocks = []

for line in open("portfolio.dat"):
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float
    holding= (name
    stocks.append(

stocks.sort()

# print "Total", t
```

.sort() sorts a list "in-place"

Note: Tuples are compared element-by-element

```
('GOOG',100,490.10)
...
('AAPL',50,118.22)
```


Looping over Sequences

```
# portfolio.py

stocks = []

for line in open("portfolio.dat"):
    fields = line.split()
    name   = fields[0]
    shares = int(fields[
    price  = float(field
    holding= (name,share
    stocks.append(holdin

stocks.sort()
for s in stocks:
    print "%-10s %8d %10.2f" % s

# print "Total", total
```

for statement iterates over any object that looks like a sequence (list, tuple, file, etc.)

Formatted I/O (again)

```
# portfolio.py

stocks = []

for line in open("portfolio.dat"):
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float(fields[2])
    holding= (name,shares,p
    stocks.append(holding)

stocks.sort()
for s in stocks:
    print "%-10s %8d %10.2f" % s

# pri
```

On each iteration, s is a tuple (name,shares,price)

```
s = ('IBM',50,91.10)
```


Calculating a Total

```
# portfolio.py

stocks = []

for line in open("portfolio.dat"):
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float(fields[2])
    holding= (nam
    stocks.append

stocks.sort()
for s in stocks:
    print "%-10s

total = sum([s[1]*s[2] for s in stocks])
print "Total", total
```

Calculate the total value of the portfolio by summing shares*price across all of the stocks

Sequence Reductions

```
# portfolio.py

stocks = []

for line in open("portfolio.dat"):
    fields = line.split()
    name
    shares
    price
    holding
    stocks.

stocks.sort
for s in st
    print "%-10s %8d %10.2f" % s

total = sum([s[1]*s[2] for s in stocks])
print "Total", total
```

Useful functions for reducing data:

sum(s) - Sums items in a sequence
min(s) - Min value in a sequence
max(s) - Max value in a sequence

List Creation

```
stocks = [                      [s[1]*s[2] for s in stocks] = [
   ('IBM',50,91.10),                      50*91.10,
   ('MSFT',200,51.23),                    200*51.23,
   ('GOOG',100,490.10),     ------------> 100*490.10,
   ('AAPL',50,118.22),                    50*118.22,
   ('SCOX',500,2.14),                     500*2.14,
   ('RHT',60,23.45)                       60*23.45
]                                       ]
```

```
    stocks.append(holding)

stocks.sort()
for s in stocks:
    print "%-10s %

total = sum([s[1]*s[2] for s in stocks])
print "Total", total
```

This operation creates a new list.
(known as a "list comprehension")

Finished Solution

```
# portfolio.py

stocks = []

for line in open("portfolio.dat"):
    fields = line.split()
    name   = fields[0]
    shares = int(fields[1])
    price  = float(fields[2])
    holding= (name,shares,price)
    stocks.append(holding)

stocks.sort()
for s in stocks:
    print "%-10s %8d %10.2f" % s

total = sum([s[1]*s[2] for s in stocks])
print "Total", total
```


Sample Output

```
shell % python portfolio.py
AAPL            50     118.22
GOOG           100     490.10
IBM             50      91.10
MSFT           200      51.23
RHT             60      23.45
SCOX           500       2.14
Total 72199.0
shell %
```

Interlude: List Processing

- Python is very adept at processing lists
- Any object can be placed in a list
- List comprehensions process list data

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

- This is shorthand for this code:

```
a = []
for i in x:
    a.append(2*i)
```


Interlude: List Filtering

- List comprehensions with a predicate

```
>>> x = [1, 2, -3, 4, -5]
>>> a = [2*i for i in x if i > 0]
>>> a
[2, 4, 8]
>>>
```

- This is shorthand for this code:

```
a = []
for i in x:
    if i > 0:
        a.append(2*i)
```

Interlude: List Comp.

- General form of list comprehensions

```
a = [expression for i in s
                   for j in t
                       ...
                   if condition ]
```

- Which is shorthand for this:

```
a = []
for i in s:
    for j in t:
        ...
            if condition:
                a.append(expression)
```


Historical Digression

- List comprehensions come from Haskell

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

a = [x*x | x <- s, x > 0]        # Haskell
```

- And this is motivated by sets (from math)

$$a = \{ x^2 \mid x \in s,\ x > 0 \}$$

- But most Python programmers would probably just view this as a "cool shortcut"

Big Idea: Being Declarative

- List comprehensions encourage a more "declarative" style of programming when processing sequences of data.
- Data can be manipulated by simply "declaring" a series of statements that perform various operations on it.

A Declarative Example

```
# portfolio.py

lines  = open("portfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

stocks.sort()
for s in stocks:
    print "%-10s %8d %10.2f" % s

total = sum([s[1]*s[2] for s in stocks])
print "Total", total
```

Files as a Sequence

```
# portfolio.py

lines  = open("portfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

stocks.sort()
for s in stocks:
    print "%-10s
total = sum([s[1]
print "Total", to
```

files are sequences of lines

```
'IBM     50     91.1\n'
'MSFT   200     51.23\n'
...
```


A List of Fields

```
# portfolio.py

lines  = open("portfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]
```

This statement creates a list of string fields

```
'IBM     50     91.10\n'        [['IBM','50',91.10'],
'MSFT   200     51.23\n'  ---->  ['MSFT','200','51.23'],
...                               ...
                                 ]
```

A List of Tuples

```
# portfolio.py

lines  = open("portfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

stocks.sort()
```

This creates a list of tuples with fields converted to numeric values

```
[['IBM','50',91.10'],          [('IBM',50,91.10),
 ['MSFT','200','51.23'],  ⟶     ('MSFT',200,51.23),
 ...                            ...
]                               ]
```


Programming Problem

- "Show me the money!"

  Dave wants to know if he can quit his day job and join a band. The file 'prices.dat' has a list of stock names and current share prices. Use it to find out.

- Write a program that reads Dave's portfolio, the file of current stock prices, and computes the gain/loss of his portfolio.
- (Oh yeah, and be "declarative")

Input Files

- portfolio.dat

```
IBM      50     91.10
MSFT    200     51.23
GOOG    100    490.10
AAPL     50    118.22
YHOO     75     28.34
SCOX    500      2.14
RHT      60     23.45
```

- prices.dat

```
IBM,117.88
MSFT,28.48
GE,38.75
CAT,75.54
GOOG,527.80
AA,36.48
SCOX,0.63
RHT,19.56
AAPL,136.76
YHOO,24.10
```


Reading Data

```
# portvalue.py

# Read the stocks in Dave's portfolio
lines  = open("portfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

# Read the current stock prices
lines  = open("prices.dat")
fields = [line.split(',') for line in lines]
prices = [(f[0],float(f[1])) for f in fields]
```

- This is using the same trick we just saw in the last section

Data Structures

```
# portvalue.py

# Read the stocks in Dave's portfolio
lines  = open("portfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

# Read the current stock prices
lines  = open("prices.dat")
fields = [line.split(',') for line in lines]
prices = [(f[0],float(f[1])) for f in fields]
```

```
stocks = [                    prices = [
  ('IBM',50,91.10),             ('IBM',117.88),
  ('MSFT',200,51.23),           ('MSFT',28.48),
  ...                           ('GE',38.75),
 ]                              ...
                               ]
```


Some Calculations

```
# portvalue.py

# Read the stocks in Dave's portfolio
lines  = open("portfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

# Read the current stock prices
lines  = open("prices.dat")
fields = [line.split(',') for line in lines]
prices = [(f[0],float(f[1])) for f in fields]

initial_value = sum([s[1]*s[2] for s in stocks])
current_value = sum([s[1]*p[1] for s in stocks
                                for p in prices
                                    if s[0] == p[0]])

print "Gain", current_value - initial_value
```

Some Calculations

```
# portvalue.py

# Read the stocks in Dave's portfolio

stocks = [                      prices = [
 ('IBM',50,91.10),               ('IBM',117.88),
 ('MSFT',200,51.23),             ('MSFT',28.48),
 ...                             ('GE',38.75),
]                                ...
                                ]

initial_value = sum([s[1]*s[2] for s in stocks])
current_value = sum([s[1]*p[1] for s in stocks
                                   for p in prices
                                       if s[0] == p[0]])

print "Gain", current_value - initial_value
```


Some Calculations

```
# portvalue.py

# Read the stocks in Dave's portfolio

stocks = [                      prices = [
 ('IBM',50,91.10),               ('IBM',117.88),
 ('MSFT',200,51.23),             ('MSFT',28.48),
 ...                             ('GE',38.75),
]                                ...
                                ]

initial_value = sum([s[1]*s[2] for s in stocks])
current_value = sum([s[1]*p[1] for s in stocks
                                   for p in prices
                                       if s[0] == p[0]])

print "Gain", current_value - initial_value
```

Some Calculations

```
# portvalue.py

# Read the stocks in Dave's portfolio

stocks = [                      prices = [
 ('IBM',50,91.10),               ('IBM',117.88),
 ('MSFT',200,51.23),             ('MSFT',28.48),
 ...                             ('GE',38.75),
]                                ...
                                ]

initial_value = sum([s[1]*s[2] for s in stocks])
current_value = sum([s[1]*p[1] for s in stocks
                                   for p in prices
                                       if s[0] == p[0]])

print "Gain", current_value - initial_value
```


Some Calculations

```
# portvalue.py

# Read the stocks in Dave's portfolio

stocks = [                      prices = [
 ('IBM',50,91.10),               ('IBM',117.88),
 ('MSFT',200,51.23),             ('MSFT',28.48),
 ...                             ('GE',38.75),
]                                ...
                                ]

initial_value = sum([s[1]*s[2] for s in stocks])
current_value = sum([s[1]*p[1] for s in stocks
                                   for p in prices
                                       if s[0] == p[0]])

print "Gain", current_value - initial_value
```

Joining two lists on a common field

Commentary

- The similarity between list comprehensions and database queries in SQL is striking
- Both are operating on sequences of data (items in a list, rows in a database table).
- If you are familiar with databases, list processing operations in Python are somewhat similar.

More on Tuples

- Tuples are commonly used to store records (e.g., rows in a database)

```
t = ('IBM', 50, 91.10)
```

- You can access elements by index

```
t[0] ⟶ 'IBM'
t[1] ⟶ 50
t[2] ⟶ 91.10
```

- You can also expand a tuple to variables

```
name, shares, price = t

name ⟶ 'IBM'
shares ⟶ 50
price ⟶ 91.10
```

Tuples and Iteration

- Tuple expansion in for-loops

```
stocks = [('IBM', 50, 91.10),
          ('MSFT',200, 51.23),
          ...
         ]

total = 0.0
for name, shares, price in stocks:
     total += shares*price
```

- This can help clarify some code

Tuples and Iteration

- Example of code with tuple expansion

```
initial = sum([shares*price
                for name, shares, price in stocks])

current = sum([s_shares*p_price
                for s_name, s_shares, s_price in stocks
                    for p_name, p_price in prices
                        if s_name == p_name])

print "Gain", current - initial
```

Iteration over multiple lists

- zip() function

```
names = ['IBM','AAPL','GOOG','YHOO','RHT']
shares = [50,50,100,20,60]

for name, nshares in zip(names,shares):
    # name = 'IBM', nshares = 50
    # name = 'AAPL',nshares = 50
    # name = 'GOOG',nshares = 100
    ...
```

- zip() creates a list of tuples

```
names  = ['IBM','AAPL','GOOG','YHOO','RHT']
shares = [50,50,100,20,60]

x = zip(names,shares)
# x = [('IBM',50),('AAPL',50),('GOOG',100),...]
```


Iteration with a counter

- enumerate() function

```
names = ['IBM','AAPL','GOOG','YHOO','RHT']
for i,n in enumerate(names):
     # i = 0, n = 'IBM'
     # i = 1, n = 'AAPL'
     # ...
```

- Example: Reading a file with line numbers

```
for linenum,line in enumerate(open("filename")):
    ...
```

Programming Problem

- Dave's Hedge Fund

  After an early morning coffee binge, Dave remembers his 'get rich' scheme and hacks up a quick Python program to automatically trade stocks before leaving to go on his morning bike ride. Upon return, he finds that his program has made 1,000,000 stock purchases, but no trades!!

- Problem: Find out how many hours Dave will have to work trimming hedges at $7/hour to pay for all of these stocks.

The Input File

- Input file: bigportfolio.dat

```
AXP 30 62.38
BA 15 98.31
DD 30 50.60
CAT 10 77.99
AIG 5 71.26
UTX 5 69.71
HD 25 37.62
IBM 20 102.77
... continues for 1000098 total lines ...
```

- Total file size: 12534017 bytes (~12 MB)

hedge.py

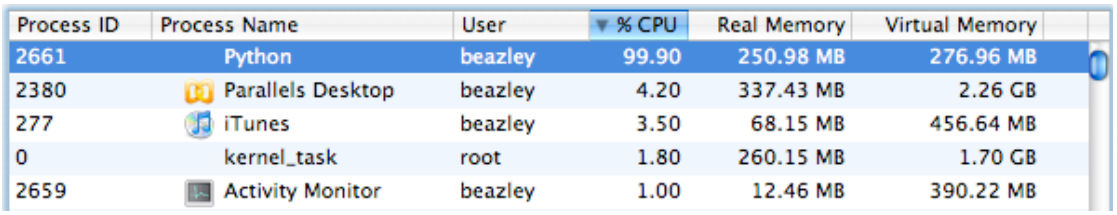
```
# hedge.py

lines  = open("bigportfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

total = sum([s[1]*s[2] for s in stocks])
print "Total", total
print "Hours of hedge clipping", total/7
```

- Output:

```
% python hedge.py
Total 1037156063.55
Hours of hedge trimming 148165151.936
```


Problem: Memory

- Our solution takes a LOT of memory

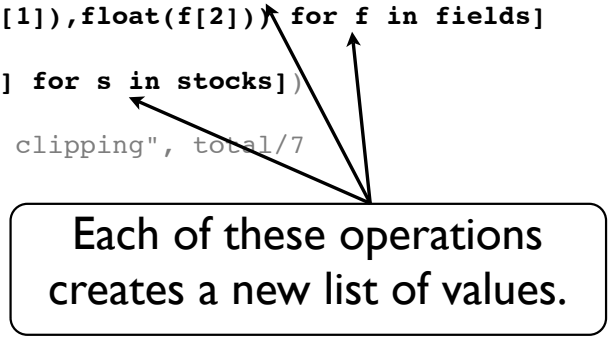


Process ID	Process Name	User	% CPU	Real Memory	Virtual Memory
2661	Python	beazley	99.90	250.98 MB	276.96 MB
2380	Parallels Desktop	beazley	4.20	337.43 MB	2.26 GB
277	iTunes	beazley	3.50	68.15 MB	456.64 MB
0	kernel_task	root	1.80	260.15 MB	1.70 GB
2659	Activity Monitor	beazley	1.00	12.46 MB	390.22 MB

- The program is constructing several large lists

Temporary Lists

```
# hedge.py

lines  = open("bigportfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

total = sum([s[1]*s[2] for s in stocks])
print "Total", total
print "Hours of hedge clipping", total/7
```



hedge2.py (2nd Attempt)

```
# hedge2.py

total = 0.0
for line in open("bigportfolio.dat"):
    fields = line.split()
    shares = int(fields[1])
    price  = float(fields[2])
    total += shares*price

print "Total", total
print "Hours of hedge trimming", total/7.00
```

- This doesn't create any lists
- But we also lose the hip "declarative" style

An Observation

- Sometimes lists are constructed as a one-time operation. Never to be used again!

```
# hedge.py

lines  = open("bigportfolio.dat")
fields = [line.split() for line in lines]
stocks = [(f[0],int(f[1]),float(f[2]) for f in fields]

total = sum([s[1]*s[2] for s in stocks])
print "Total", total
print "Hours of hedge clipping", total/7
```

- Notice in this code: data in fields, stocks, and sum() is only used once.

Generated Sequences

- Generator expressions

```
x = [1,2,3,4]
y = (i*i for i in x)
```

- Creates an object that generates values when iterating (which only works once)

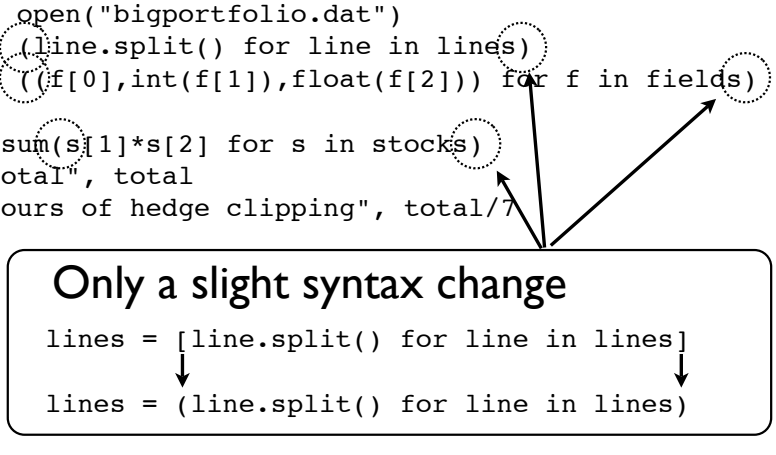
```
>>> y
<generator object at 0x6e378>
>>> for a in y: print a
...
1
4
9
16
>>> for a in y: print a
...
>>>
```

hedge3.py (3rd Attempt)

```
# hedge3.py

lines  = open("bigportfolio.dat")
fields = (line.split() for line in lines)
stocks = ((f[0],int(f[1]),float(f[2])) for f in fields)

total = sum(s[1]*s[2] for s in stocks)
print "Total", total
print "Hours of hedge clipping", total/7
```


A Generated Solution

```
# hedge3.py

lines  = open("bigportfolio.dat")
fields = (line.split() for line in lines)
stocks = ((f[0],int(f[1]),float(f[2])) for f in fields)

total = sum(s[1]*s[2] for s in stocks)
print "Total", total
print "Hours of hedge clipping", total/7
```



A Generated Solution

```
# hedge3.py

lines  = open("bigportfolio.dat")
fields = (line.split() for line in lines)
stocks = ((f[0],int(f[1]),float(f[2])) for f in fields)

total = sum(s[1]*s[2] for s in stocks)
print "Total", total
print "Hours of hedge clipping", total/7
```

For functions that operate on sequences, you can generate the sequence in the function argument (the syntax looks a little exotic).

Running the Solution

- It works!

```
shell % python hedge3.py
Total 1037156063.55
Hours of hedge trimming 148165151.936
shell %
```

- And it uses very little memory!

Process ID	Process Name	User	% CPU	Real Memory	Virtual Memory
2685	Python	beazley	99.90	2.56 MB	36.14 MB
68	WindowServer	windowserve	9.10	151.10 MB	802.21 MB
2380	Parallels Desktop	beazley	4.20	337.50 MB	2.26 GB
277	iTunes	beazley	2.50	67.70 MB	455.01 MB

- And it runs about 3x faster than before

Interlude : Data Processing

- So far, we've used Python to process data
- And we used a lot of advanced machinery
  - List comprehensions
  - Generator Expressions
  - Programming in a "declarative" style
- Question : Is Python an appropriate tool??
  - What is the performance?

Python vs. Awk

- Let's put it head-to-head

```
{  total += $2 * $3 } END {
  print "Total", total
  print "Hours of hedge trimming", total/7
}
```

- Performance (bigportfolio.dat)

```
AWK     : 1.03 seconds
Python  : 2.25 seconds
```

- Memory (bigportfolio.dat)

```
AWK     : 516 KB
Python  : 2560 KB
```

- System Notes: Mac Pro (2x2.66 Ghz Dual Core Intel Xeon)

Commentary

- It's not surprising that Python is slower than AWK. It's a much more complex language.
- However, it's not slow enough to make me lose a lot of sleep about it.
- Your mileage may vary.

Segue: Ordered Data

- All examples have used "ordered" data
  - Sequence of lines in a file
  - Sequence of fields in a line
  - Sequence of stocks in a portfolio
- What about unordered data?

Dictionaries

- A hash table or associative array
- Example: A table of stock prices

```
prices = {
   'IBM'  : 117.88,
   'MSFT' : 28.48,
   'GE'   : 38.75,
   'CAT'  : 75.54,
   'GOOG' : 527.80
}
```

- Allows random access using key names

```
>>> prices['GE']              # Lookup
38.75
>>> prices['GOOG'] = 528.50   # Assignment
>>>
```


Dictionaries

- Dictionaries as a data structure
- Named fields

```
stock = {
   'name'   : 'GOOG',
   'shares' : 100,
   'price'  : 490.10
}
```

- Example use

```
>>> cost = stock['shares']*stock['price']
>>> cost
49010.0
>>>
```

Programming Problem

- "Show me the money!" - Part Deux

  Dave wants to know if he can quit his day job and join a band. The file 'prices.dat' has a list of stock names and current share prices. Use it to find out.

- Write a program that reads Dave's portfolio, the file of current stock prices, and computes the gain/loss of his portfolio.
- Use dictionaries

Solution : Part I

- Creating a list of stocks in the portfolio

```
# portvalue2.py
# Compute the value of Dave's portfolio

stocks = []
for line in open("portfolio.dat"):
    fields = line.split()
    record = {
        'name'   : fields[0],
        'shares' : int(fields[1]),
        'price'  : float(fields[2])
    }
    stocks.append(record)
```

Dictionary Data Structures

```
# portvalue2.py
# Compute the value of Dave's portfolio

stocks = []
for line in open("portfolio.dat"):
    fields = line.split()
    record = {
        'name'   : fields[0],
        'shares' : int(fields[1]),
        'price'  : float(fields[2])
    }
    stocks.append(record)
```

Each stock is a dict

```
record = {
  'name'   : 'IBM',
  'shares' : 50
  'price'  : 91.10
}
```


Lists of Dictionaries

- A list of objects with "named fields."

```
# portvalue2.py
# Compute the value of Dave's

stocks = []
for line in open("portfolio.da
    fields = line.split()
    record = {
        'name'   : fields[0],
        'shares' : int(fields[
        'price'  : float(field
    }
    stocks.append(record)
```

```
stocks = [
   {'name'   :'IBM',
    'shares' : 50,
    'price'  : 91.10 },
   {'name'   :'MSFT',
    'shares' : 200,
    'price'  : 51.23 },
   ...
]
```

Example:

```
stocks[1] → {'name'   : 'MSFT',
             'shares' : 200,
             'price'  : 51.23}

stocks[1]['shares'] → 200
```

Solution : Part 2

- Creating a dictionary of current prices

```
prices = {}
for line in open("prices.dat"):
    fields = line.split(',')
    prices[fields[0]] = float(fields[1])
```

- Example:

```
prices   {
   'GE'   : 38.75,
   'AA'   : 36.48,
   'IBM'  : 117.88,
   'AAPL' : 136.76,
   ...
}
```


Solution : Part 3

- Calculating portfolio value and gain

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

current = sum(s['shares']*prices[s['name']]
                  for s in stocks)

print "Current value", current
print "Gain", current - initial
```

- Note: Using generator expressions

Solution : Part 3

- Calculating portfolio value and gain

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

current = sum(s['shares']*prices[s['name']]
              for s in stocks)

print "Current value", current
print "Gain", current - initial
```

Fast price lookup

```
s = {
  'name'   : 'IBM',
  'shares' : 50
  'price'  : 91.10
}
```

```
prices   {
   'GE'   : 38.75,
   'AA'   : 36.48,
   'IBM'  : 117.88,
   'AAPL' : 136.76,
   ...
}
```


More on Dictionaries

- Getting an item

```
x = prices['IBM']
y = prices.get('IBM',0.0)  # w/default if not found
```

- Adding or modifying an item

```
prices['AAPL'] = 145.14
```

- Deleting an item

```
del prices['SCOX']
```

- Membership test (in operator)

```
if 'GOOG' in prices:
    x = prices['GOOG']
```

More on Dictionaries

- Number of items in a dictionary

```
n = len(prices)
```

- Getting a list of all keys (unordered)

```
names = list(prices)
names = prices.keys()
```

- Getting a list of all values (unordered)

```
prices = prices.values()
```

- Getting a list of (key,value) tuples

```
data = prices.items()
```


The Story So Far

- Primitive data types: Integers, Floats, Strings
- Compound data: Tuples
- Sequence data: Lists
- Unordered data: Dictionaries

The Story So Far

- Powerful support for iteration
- Useful data processing primitives (list comprehensions, generator expressions)
- Bottom line:

  Significant tasks can be accomplished doing nothing more than manipulating simple Python objects (lists, tuples, dicts)

Remaining Topics

- Details on Python object model
- Errors and exception handling
- Functions
- Modules
- Classes and objects

Object Mutability

- Objects fall into two categories
  - Immutable (can't be changed)
  - Mutable (can be changed)
- Mutable: Lists, Dictionaries
- Immutable: Numbers, strings, tuples
- All of this ties into memory management (which is why we would care about such a seemingly low-level implementation detail)

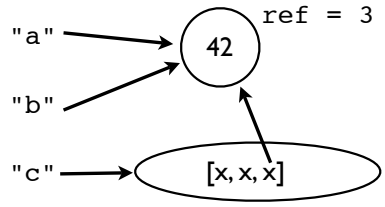
Variable Assignment

- Variables in Python are only names
- Assignment does not store a value into a fixed memory location (like C)
- It is only a name assignment to an object

Reference Counting

- Objects are reference counted
- Increased by assignment, inclusion

```
a = 42
b = a
c = [1,2]
c.append(b)
```



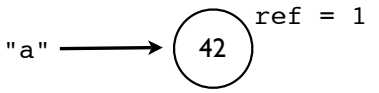
- Can check using the is operator

```
>>> a is b
True
>>> a is c[2]
True
```

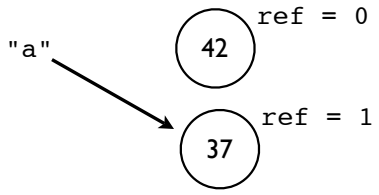

Reference Counting

- Important point: assignment does not copy!

```
a = 42
```




```
a = 37
```

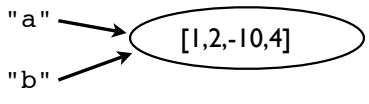


- Creates a new object
- Makes the name refer to it

Reference Counting

- Common pitfall: "duplicating" a container

```
>>> a = [1,2,3,4]
>>> b = a
>>> b[2] = -10
>>> a
[1,2,-10,4]
```



- Other techniques must be used for copying

```
>>> a = [1,2,3,4]
>>> b = list(a)       # Create a new list from a
>>> b[2] = -10
>>> a
[1,2,3,4]
```

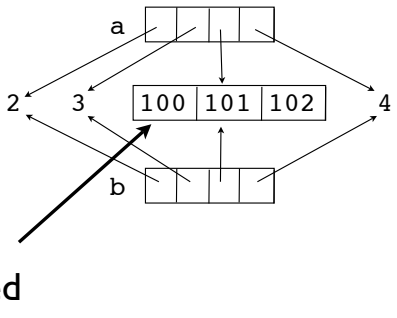

Shallow Copies

- Creating a new list only makes a shallow copy

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

- However, items in list copied by reference

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



Deep Copying

- Makes a copy of an object and copies 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]
>>>
```


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 example

- These functions do data conversions

```
int(x)
float(x)
str(x)
```

- Let's put them in a list

```
fieldtypes = [str, int, float]
```

- Let's make some tuples

```
fields = ['GOOG','100','490.10']
typed_fields = zip(fieldtypes,fields)
# [(str,'GOOG'),(int,'100'),(float,490.10)]
```

- Let's make values

```
values = [ty(field) for ty,field in typed_fields]
# values = ['GOOG',100,490.10]
```


First-class Commentary

- The fact that all objects are first-class may take some time to sink in.
- Especially if you come from C/C++
- Can be used for very compact, interesting styles of programming.
- All named program elements can be treated as data and used in surprising ways.

Object type

- All objects have a type

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

- type() function will tell you what it is
- Typename usually a constructor function

```
>>> str(42)
'42'
>>>
```


Type Checking

- How to tell if an object is a specific type

```
if type(a) is list:
    print "a is a list"

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

- Checking for one of many types

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

Exceptions

- In Python, errors are reported as exceptions
- Causes the program to stop
- Example:

```
>>> prices = { 'IBM' : 91.10,
...            'GOOG' : 490.10 }
>>> prices['SCOX']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'SCOX'
>>>
```

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

Exceptions

- Exceptions can be caught
- To catch, use try-except

```
try:
    print prices["SCOX"]
except KeyError:
    print "No such name"
```

- To raise an exception, use raise

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


Exceptions

- Code can specify actions that must always run

```
f = open(filename,"r")
try:
    ...
finally:
    f.close()    # Runs regardless of exception
```

- finally block runs regardless of whether or not an exception occurred
- Typically used to properly manage resources

Program Organization

- Python provides a few basic primitives for structuring larger programs
  - Functions
  - Modules
  - Classes
- Will use these as programs grow in size

Functions

- Defined with the def statement

```
def read_portfolio(filename):
    stocks = []
    for line in open(filename):
         fields = line.split()
         record = { 'name' : fields[0],
                    'shares' : int(fields[1]),
                    'price' : float(fields[2]) }
         stocks.append(record)
    return stocks
```

- Using a function

```
stocks = read_portfolio('portfolio.dat')
```

Function Examples

```
# Read prices into a dictionary
def read_prices(filename):
    prices = { }
    for line in open(filename):
         fields = line.split(',')
         prices[fields[0]] = float(fields[1])
    return prices

# Calculate current value of a portfolio
def portfolio_value(stocks,prices):
    return sum(s['shares']*prices[s['name']]
                  for s in stocks)
```


Function Examples

- A program that uses our functions

```
# Calculate the value of Dave's portfolio

stocks = read_portfolio("portfolio.dat")
prices = read_prices("prices.dat")
value  = portfolio_value(stocks,prices)

print "Current value", value
```

- Commentary: There are no major surprises with functions--they work like you would expect.

Generator Functions

- A function that generates values (using yield)
- The primary use is with iteration (for-loop)

```
def make_fields(lines,delimeter=None):
    for line in lines:
         fields = line.split(delimeter)
         yield fields
```

- Big idea: this function will generate a sequence of values one at a time instead of returning results all at once.
- Generation of values continues until return

Using a Generator Func

- Generator functions almost always used in conjunction with the for statement

```
fields = make_fields(open("portfolio.dat"))
stocks = [(f[0],int(f[1]),float(f[2])) for f in fields]

fields = make_fields(open("prices.dat"),',')
prices = {}
for f in fields:
    prices[f[0]] = float(f[1])
```

- On each iteration of the for-loop, the yield statement produces a new value. Looping stops when the generator function returns

Modules

- As programs grow, you will want multiple source files
- Also to re-use previous code
- Any Python source file is a module
- Just use the import statement

A Sample Module

```
# stockfunc.py

def read_portfolio(filename):
    lines = open(filename)
    fields = make_fields(lines)
    return [ { 'name' : f[0],
               'shares' : int(f[1]),
               'price'  : float(f[2]) } for f in fields]
# Read prices into a dictionary
def read_prices(filename):
    prices = { }
    for line in open(filename):
         fields = line.split(',')
         prices[fields[0]] = float(fields[1])
    return prices

# Calculate current value of a portfolio
def portfolio_value(stocks,prices):
    return sum(s['shares']*prices[s['name']]
                  for s in stocks)
```

Using a Module

- importing a module

```
import stockfunc

stocks = stockfunc.read_portfolio("portfolio.dat")
prices = stockfunc.read_prices("prices.dat")
value  = stockfunc.portfolio_value(stocks,prices)
```

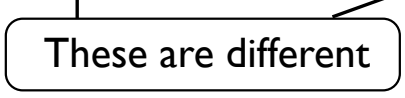
- Modules define namespaces
- All contents accessed through module name

Modules as Namespaces

- All objects in your program always live inside some module.
- For global variables, we're really talking about variables at the module level.

```
# foo.py
x = 42
```

```
# bar.py
x = "Hello World"
```




```
>>> import foo
>>> import bar
>>> foo.x
42
>>> bar.x
'Hello World'
>>>
```

from module import

- Symbols from a module can be imported into the namespace of another module

```
from stockfunc import read_portfolio

stocks = read_portfolio("portfolio.dat")
```

- Importing all symbols

```
from stockfunc import *

stocks = read_portfolio("portfolio.dat")
prices = read_prices("prices.dat")
value  = portfolio_value(stocks,prices)
```

- This is only an export of symbol names. The code in the imported module still runs in its own module namespace however.

Python Standard Library

- Python comes with several hundred modules
  - Text processing/parsing
  - Files and I/O
  - Systems programming
  - Network programming
  - Internet
  - Standard data formats
- Will cover some of these in afternoon section

A Few Critical Modules

- Modules that are used quite frequently
  - sys. Command line options, standard I/O
  - math. Math functions (sqrt, sin, cos, etc.)
  - copy. Copying of objects
  - re. Regular expressions
  - os. Operating system functions.

Classes and Objects

- Python provides full support for objects
- Defined with the class statement

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
```

Using an Object

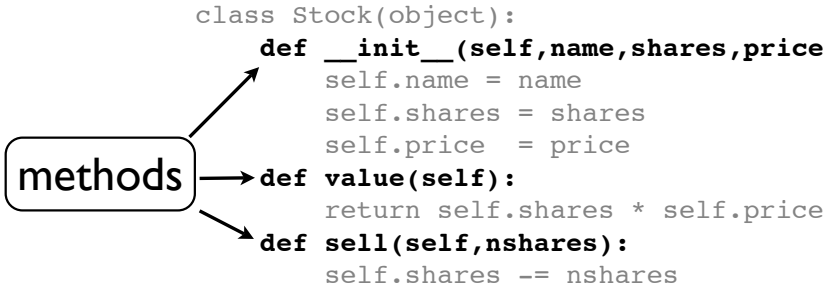
- Creating an object and calling methods

```
>>> s = Stock('GOOG',100,490.10)
>>> s.name
'GOOG'
>>> s.shares
100
>>> s.value()
49010.0
>>> s.sell(25)
>>> s.shares
75
```

- Basically, an object is just a way to package data and functions together.

Classes and Methods

- A class is a just a collection of "methods"
- A method is just a function




```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
```

Methods and Instances

- Methods always operate on an "instance"
- Passed as the first argument (self)

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
```

Creating Instances

- Class used as a function to create instances
- This calls __init__() (Initializer)

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
```

```
>>> s = Stock('GOOG',100,490.10)
>>> print s
<__main__.Stock object at 0x6b910>
>>>
```

Instance Data

- Each instance holds data (state)
- Created by assigning attributes on self

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
```

Instance data

```
>>> s = Stock('GOOG',100,490.10)
>>> s.name
'GOOG'
>>> s.shares
100
>>>
```


Calling Methods

- Methods are invoked on an instance
- Instance is passed as first parameter

```
class Stock(object):
    def __init__(self,name,shares,price):
        self.name = name
        self.shares = shares
        self.price  = price
    def value(self):
        return self.shares * self.price
    def sell(self,nshares):
        self.shares -= nshares
```

```
>>> s = Stock('GOOG',100,490.10)
>>> s.value()
49010.0
>>> s.sell(50)
>>>
```

Object Commentary

- There is much more to objects in Python
- However, I made a conscious decision not to make objects the primary focus of this tutorial.
- We will use some simple classes later, but I won't be going to more detail on how classes work or some of their more advanced features.

Historical Note

- Classes were one of the last features added to Python when it was first created (almost as an afterthought).
- Although knowing about classes is important, Python does not treat them as a cultish religion or the one true path to enlightenment.
- You can write very powerful programs without using classes much at all (will see later)

The End of the Intro

- Python has a small set of very useful datatypes (numbers, strings, tuples, lists, and dictionaries)
- There are very powerful operations for manipulating data
- Programs can be organized using functions, modules, and classes
- This is the essential information you need to know to get started.