Python Mastery

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

Edition: Fri Mar 11 11:43:26 2016

Python Mastery : Table of Contents

Edition: Fri Mar 11 11:43:26 2016

Course Summary

Python Mastery is an accelerated training course designed for programmers who already know the basics of Python, but who want to know how to use it more effectively. A major focus of the course is on advanced features that are widely used in the Python world, but which don't get much coverage in more introductory tutorials and books. This includes the inner workings of the Python object system, as well as decorators, metaclasses, generators, coroutines, closures, and context managers.

System Requirements

The course assumes the use of Python 3.5 on any operating system platform.

Support Files and Exercises

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

http://www.dabeaz.com/python/pythonmaster.zip

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

0. Course Setup

1. Python Review

2. Idiomatic Data Handling

Section 0

Course Setup

Required Files

- Where to get Python (if not installed)

http://www.python.org

- This course is written for Python 3.5
- Almost everything also applies to Python 2
- Exercises for this class

http://www.dabeaz.com/python/pythonmaster.zip

Working Environment

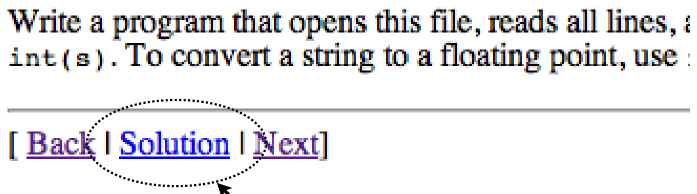
- This is not an introductory course
- Use whatever tools you currently use to develop Python code
- Editors, IDEs, etc.
- Almost everything in this course is platform-neutral and will work everywhere

Class Exercises

- Exercise descriptions are found in

  pythonmaster/Exercises/index.html

- All exercises have solution code


Write a program that opens this file, reads all lines,
int(s). To convert a string to a floating point, use

[Back | Solution | Next]

Look for the link at the bottom!

Class Exercises

- Working solution code can be found in

  pythonmaster/Solutions

- Each problem has its own directory

2_1/	Exercise 2.1
2_2/	Exercise 2.2
...	

General Tips

- Try to save all of your work in "pythonmaster"
- Ask for help if stuck
- There is no shame in looking at solution code (it's fine to copy it, modify it, etc.)
- Pace yourself

Section 1

Python Review

(Optional)

Overview

- A very fast-paced review of Python
- The absolute basics that you should already know if you are taking this course
- Essential details for later parts of the class

All Things Python

http://www.python.org

- Downloads
- Documentation and tutorial
- Community Links
- News and more
- Tutorial

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 % python3
Python 3.5.0 (default, Oct 27 2015, 13:20:23)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)]
>>>
```

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

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)
- Make sure you use "python" mode

Running Programs

- Command line

```
bash % python3 helloworld.py
hello world
bash %
```

- #! (Unix)

```
#!/usr/bin/env python3
# helloworld.py
print('hello world')
```

python -i

- For debugging, use python -i

```
bash % python3 -i helloworld.py
hello world
>>>
```

- Runs your program and then enters the Python interactive shell afterwards
- Quite useful for debugging, testing, etc.

Exercise 1.1

Time: 10 minutes

Program Execution

- 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

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., /* ... */).

Variables

- A variable is a name for some value
- Variable names follow same rules as C
  [A-Z a-z_][A-Za-z0-9_]*
- You do <u>not</u> 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.

Expressions

- Math works normally (precedence, assoc, etc.)

```
a = 2 + 3
b = 2 + 3 * 4
c = (2 + 3) * 4
```

- Operators are same as C

```
+, -, *, /, %, <<, >>, &, |, ^, ...
```

- Other operators (Python-specific)

```
7 // 4              Truncating division
7 ** 4              Power operator
```

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
```

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')
```

Looping

- While statement loops on a condition

```
while count > 0:
    print('T-minus', count)
    count -= 1
print('Boom!')
```

- For-loop iterates over items (e.g., foreach)

```
nums = [1,7,10,23]
for x in nums:
    print(x)          # Prints 1, 7, 10, 23
```

Looping Control-Flow

- break - terminates a loop early

```
for name in names:
    if name == 'python':
        break
    ...
```

- continue - skip to next iteration

```
for line in lines:
    if line == '\n':   # Ignore blank lines
        continue
    ...
```

Printing

- The print function (Python 3)

```
print(x)
print(x,y,z)
print('Your name is', name)
```

- Produces a single line of text
- Items are separated by spaces
- In Python 2, print is a statement

```
print x
print x,y,z
print 'Your name is', name
```

Formatted Printing

- Use % operator

```
print('%10s %10d %10.2f' % (name, shares, price))
```

- Or .format()

```
print('{:10s} {:10d} {:10.2f}'.format(name,shares,price))
```

pass statement

- Sometimes you will need to specify an empty block of code

```
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)

Core Python Objects

- Programs are built upon a core set of built-in datatypes (numbers, strings, lists, etc.)

```
None                      # Nothing, nada, nil, ...
True                      # Boolean
23                        # Integer
12345678123901234L        # Long integer
2.3                       # Float
2+3j                      # Complex
'Hello World'             # String
u'Spicy Jalape\u00f1o'    # Unicode string
('www.python.org',80)     # Tuple
[1,2,3,4]                 # List
{'name':'IBM', ... }      # Dictionary
```

- You should have already used most of these types if you've written any Python at all

Manipulating Objects

- Objects are manipulated by operators

```
a + b                     # Add
x = a[i]                  # Indexed lookup
y = a[i:j]                # Slicing
a[i] = val                # Indexed assignment
x in a                    # Containment
... many others ...
```

- Also manipulated by various methods

```
a.find('python')
a.split(',')
b.append(2)
...
```

- Available operators/methods depends on the object being manipulated

Exercise 1.2

Time: 15 minutes

File Input and Output

- Opening a file

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

- To read data

```
line = f.readline()          # Read a line of text
data = f.read([maxbytes])    # Read data
```

- To write text to a file

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

- Closing a file (when done)

```
f.close()
```

Common Idioms

- Reading a file line-by-line

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

- Reading an entire file into a string

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

Closing Files

- Files need to be closed after use

```
f = open('foo.txt','r')
# Use f
...
f.close()
```

- In modern Python, use 'with'

```
with open('foo.txt','r') as f:
    # Use f
    ...
# Automatically closed here
```

Text Data

- When reading text, Python 3 assumes unicode
- You might need to give an encoding

```
f = open('foo.txt','r',encoding='latin-1')
```

- For Unicode in Python 2, use io module

```
import io
f = io.open('foo.txt', 'r', encoding='latin-1')
```

- We don't actually cover much unicode in this course (mention it only occasionally)

Exercise 1.3

Time: 10 minutes

Simple Functions

- Use functions for code you want to reuse

```
def sumcount(n):
    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

Simple Functions

- Functions behave in a sane manner
  - Inner variables have local scope
  - Things like recursion work fine
  - You can have default arguments
- Will say more about functions later, but you should already know how to define and use simple function definitions

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)

```
% python3 foo.py
Traceback (most recent call last):
  File "foo.py", line 21, in <module>
    raise RuntimeError("What a kerfuffle")
RuntimeError: What a kerfuffle
```

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 ValueError as e:
    ...
except TypeError as e:
    ...
```

- Alternatively, if handling is the same

```
try:
    ...
except (ValueError, TypeError) as e:
    ...
```

- Catching any exception (danger awaits)

```
try:
    ...
except Exception as e:
    ...
```

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.)

Exercise 1.4

Time: 10 minutes

Objects

- Python is an object-oriented language
- All of the basic data types (integers, strings, lists, etc.) are examples of "objects"
- Objects involve data and a set of "methods" that carry out various operations

```
a = 'Hello World'     # A string object
b = a.upper()         # A method applied to the string

items = [1,2,3]       # A list object
items.append(4)       # A method applied to the list
```

Classes

- You can make your own 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 all of the function definitions that implement the various methods

Instances

- Created by calling the class as a function

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

- Each instance gets its own data

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

- Invoke the methods as follows

```
>>> 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

- Mostly, it just stores the data attributes

Methods

- Functions that operate on instances

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

- The object is just 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 the "this" pointer in C++.

Exercise 1.5

Time: 10 minutes

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)

import as statement

- Changes the local name of a module

```
# bar.py
import math as m

a = m.sin(x)
```

- Exactly the same as import except the module object is assigned a different name
- The new name only applies locally within the source file that did the import (other files can import using the standard name without any confusion)

from module import

- Lifts selected symbols out of a module and puts them into local scope

```
# bar.py
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

from module import *

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

```
# bar.py
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

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 % python3 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

Summary

- This has been an overview of basics
- If you've already been programming Python for awhile, you should already know this material
- Later sections go into much more depth

Exercise 1.6

Time: 5 minutes

Section 2

Idiomatic Data Handling

Data Structures

- Real programs must deal with 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)

Data Structures

- Some options
  - Tuple
  - Dictionary
  - Class instance
  - Named tuple
- Let's take a short tour

Tuples

- A collection of values packed together

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

- Can use like an array

```
name = s[0]
cost = s[1] * s[2]
```

- Unpacking into separate variables

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

- Immutable

```
s[1] = 75     # TypeError. No item assignment
```

Dictionaries

- An unordered set of values indexed by "keys"

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

- Use the key name to access

```
name = s['name']
cost = s['shares'] * s['price']
```

- Modifications are allowed

```
s['shares'] = 75
s['date'] = '7/25/2015'
del s['name']
```

Dictionary Thoughts

- Dictionaries are often a good choice for representing simple data records
- Easy to manipulate (can freely change fields, modify values, etc.)
- Improved readability (key names are usually more descriptive than numeric indices)
- Easy interoperability (e.g., convert to JSON)

User-Defined Classes

- A simple data structure class

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

- This gives you the nice object syntax...

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s.name
'GOOG'
>>> s.shares * s.price
49010.0
>>>
```

Classes and Slots

- For data structures, consider adding __slots__

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

- Slots is a performance optimization that is specifically aimed at data structures
- Less memory and faster attribute access

Named Tuples

- namedtuple(*clsname*, *fieldnames*)

```
from collections import namedtuple

Stock = namedtuple('Stock',
                   ['name', 'shares', 'price'])
```

- It creates a class that you use to make instances

```
>>> s = Stock('GOOG',100,490.1)
>>> s.name
'GOOG'
>>> s.shares * s.price
49010.0
>>>
```

Named Tuples

- Named tuples retain all of the core features of tuples (immutability, unpacking, indexing, etc.)

```
>>> s = Stock('GOOG', 100, 490.1)
>>> s[0]
'GOOG'
>>> name, shares, price = s
>>> print('%10s %10d %10.2f' % s)
      GOOG        100     490.10
>>> isinstance(s, tuple)
True
>>> s.name = 'ACME'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Exercise 2.1

Time : 20 minutes

Containers

- Programs often have to work many objects
  - Lists (ordered data)
  - Sets (unordered data, no duplicates)
  - Dictionaries (unordered key-value data)
- The choice depends on the problem

Lists

- Use a list when the order of data matters
- Example: A list of tuples

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44)
]

portfolio[0] ────────→ ('GOOG', 100, 490.1)
portfolio[1] ────────→ ('IBM', 50, 91.1)
```

- Lists can be sorted and rearranged

Sets

- A set is an unordered collection of items

```
a = {'IBM','AA','AAPL' }
```

- Sets can eliminate duplicates

```
names = ['IBM','YHOO','IBM','CAT','MSFT','CAT','IBM']
unique_names = set(names)
```

- Sets are useful for membership tests

```
members = set()

members.add(item)       # Add an item
members.remove(item)    # Remove an item

if item in members:     # Test for membership
   ...
```

Dicts

- Useful for indices and lookup tables

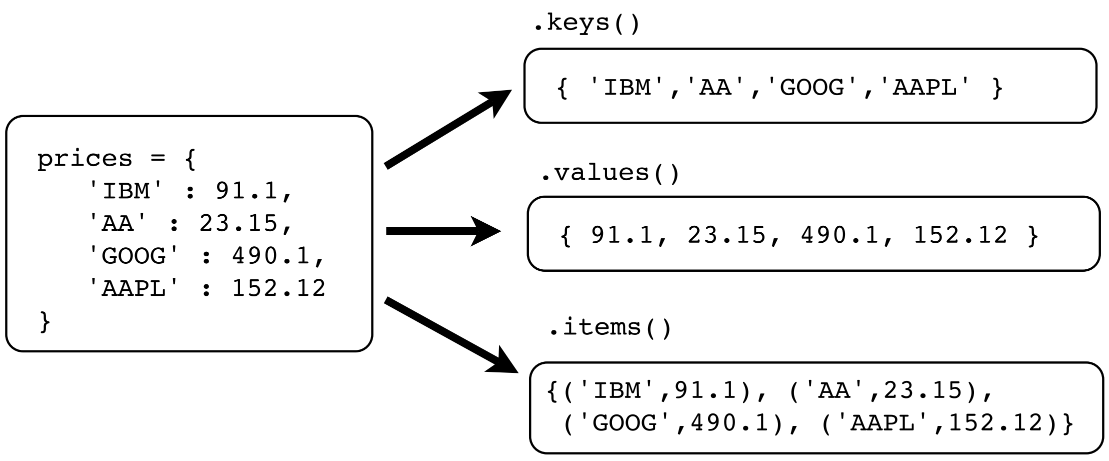
```
prices = {
    'GOOG' : 513.25,
    'CAT'  : 87.22,
    'IBM'  : 93.37,
    'MSFT' : 44.12
    ...
}
```

- Common use

```
p = prices['IBM']              # Value lookup
p = prices.get('AAPL', 0.0)    # Lookup with default value
prices['HPE'] = 37.42          # Assignment

if name in prices:             # Membership test
   ...
```

Dictionary Views

- Dict contents are sometimes viewed as three different sets of data



```
prices = {
   'IBM' : 91.1,
   'AA' : 23.15,
   'GOOG' : 490.1,
   'AAPL' : 152.12
}
```

.keys()

```
{ 'IBM','AA','GOOG','AAPL' }
```

.values()

```
{ 91.1, 23.15, 490.1, 152.12 }
```

.items()

```
{('IBM',91.1), ('AA',23.15),
 ('GOOG',490.1), ('AAPL',152.12)}
```

Dictionary Views

- Views are overlays (not copies)
- Updates to the dict are reflected in the view

```
>>> names = prices.keys()
>>> names
dict_keys(['AA', 'GOOG', 'AAPL', 'IBM'])

>>> prices['HPE'] = 42.13
>>> names
dict_keys(['AA', 'GOOG', 'AAPL', 'IBM', 'HPE'])
```

Notice new entry in the view

List/Dict Conversions

- dict(pairs) - Create a dict from key/value pairs

```
>>> data = [ ('GOOG',490.1), ('AA',23.15), ('IBM',91.5) ]
>>> dict(data)
{ 'AA': 23.15, 'IBM': 91.5, 'GOOG': 490.1 }
>>>
```

- list(dict) - Create a list of key names

```
>>> data = { 'AA': 23.15, 'IBM': 91.5, 'GOOG': 490.1 }
>>> list(data)
['AA', 'IBM', 'GOOG']
>>>
```

- list(dict.items()) - Create a list of key/value pairs

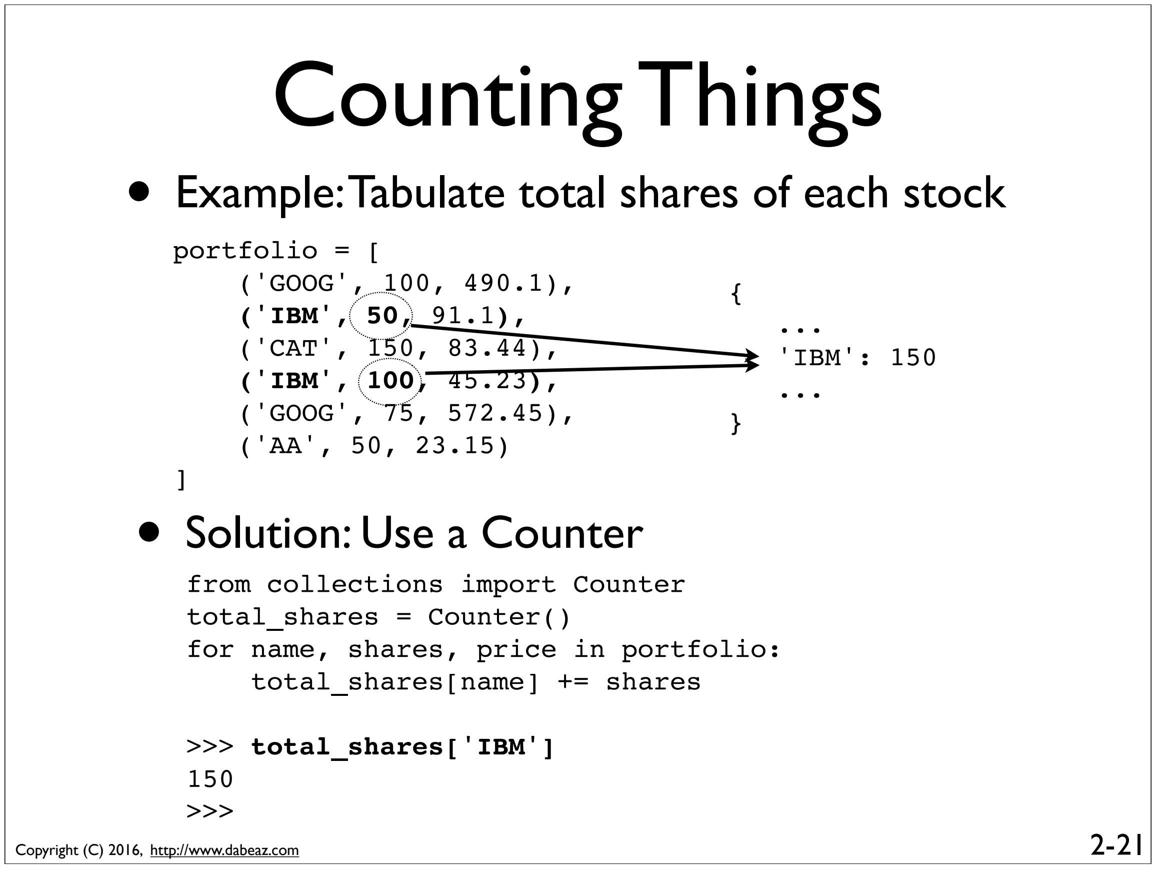
```
>>> data = { 'AA': 23.15, 'IBM': 91.5, 'GOOG': 490.1 }
>>> list(data.items())
[('AA', 23.15), ('IBM', 91.5), ('GOOG', 490.1) ]
>>>
```

Exercise 2.2

Time : 20 minutes

The Secret Containers

- The collections module
  - defaultdict
  - Counter
  - deque
  - OrderedDict
  - ... and more



Counting Things

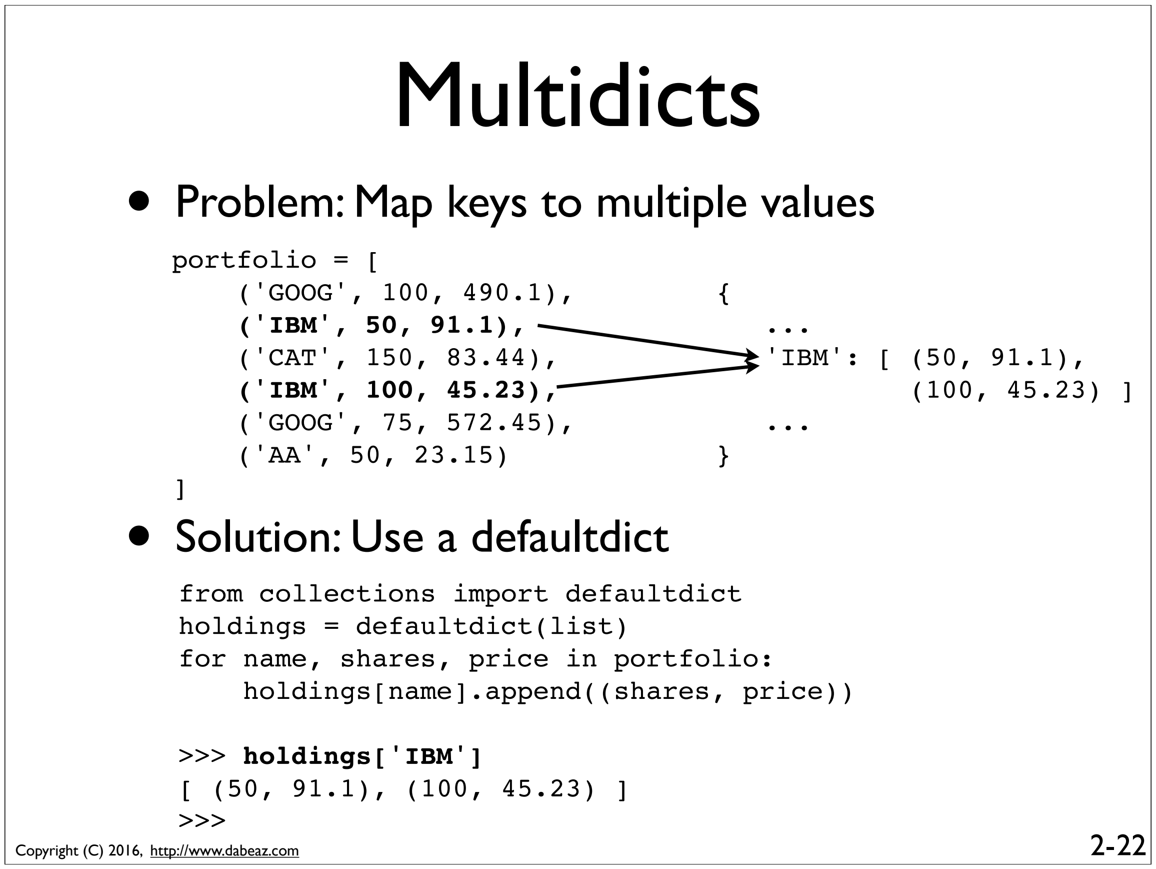
- Example: Tabulate total shares of each stock

```
portfolio = [
    ('GOOG', 100, 490.1),          {
    ('IBM', 50, 91.1),                ...
    ('CAT', 150, 83.44),              'IBM': 150
    ('IBM', 100, 45.23),              ...
    ('GOOG', 75, 572.45),          }
    ('AA', 50, 23.15)
]
```

- Solution: Use a Counter

```
from collections import Counter
total_shares = Counter()
for name, shares, price in portfolio:
    total_shares[name] += shares

>>> total_shares['IBM']
150
>>>
```

Copyright (C) 2016, http://www.dabeaz.com 2-21



Multidicts

- Problem: Map keys to multiple values

```
portfolio = [
    ('GOOG', 100, 490.1),         {
    ('IBM', 50, 91.1),               ...
    ('CAT', 150, 83.44),             'IBM': [ (50, 91.1),
    ('IBM', 100, 45.23),                      (100, 45.23) ]
    ('GOOG', 75, 572.45),            ...
    ('AA', 50, 23.15)             }
]
```

- Solution: Use a defaultdict

```
from collections import defaultdict
holdings = defaultdict(list)
for name, shares, price in portfolio:
    holdings[name].append((shares, price))

>>> holdings['IBM']
[ (50, 91.1), (100, 45.23) ]
>>>
```

Copyright (C) 2016, http://www.dabeaz.com 2-22

Keeping a History

- Problem: Keep a history of the last N things

```
line1
line2
line3
line4          history = [ line3, line4, line5 ]
line5
...
```

- Solution: Use a deque

```
from collections import deque

history = deque(maxlen=N)
with open(filename) as f:
    for line in f:
        history.append(line)
        ...
```

Controlling Dict Order

- Problem: Create a dict where keys go in the order that you want
- Solution: OrderedDict

```
from collections import OrderedDict
s = OrderedDict()
s['name'] = 'GOOG'
s['shares'] = 100
s['price'] = 490.1

>>> for key, val in s.items():
...     print(key,'=', val)
...
name = GOOG
shares = 100
price = 490.1
>>>
```

Commentary

- collections is a useful module to know
- Simplifies many common data handling problems
- If you're not using it, you're missing out

Exercise 2.3

Time : 15 minutes

Iteration

- The for-loop iterates over a sequence

```
>>> names = ['IBM', 'YHOO', 'AA', 'CAT' ]
>>> for name in names:
...     print(name)
...
IBM
YHOO
AA
CAT
>>>
```

- It seems simple enough...

Iterating on Tuples

- Consider a list of tuples

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]
```

- Iteration with unpacking

```
for name, shares, price in portfolio:
    ...
```

- Iteration with a "throwaway" value (use __)

```
for name, __, price in portfolio:
    ...
```

Iterating on Varying Records

- Consider a list of varying sized data structures

```
prices = [
    ['GOOG', 490.1, 485.25, 487.5 ],
    ['IBM', 91.5],
    ['HPE', 13.75, 12.1, 13.25, 14.2, 13.5 ],
    ['CAT', 52.5, 51.2]
]
```

- Wildcard unpacking (Python 3 only)

```
for name, *values in prices:
    print(name, values)
```

```
name    values
'GOOG'  [490.1, 485.25, 487.5 ]
'IBM'   [91.5]
'HPE'   [13.75, 12.1, 13.25, 14.2, 13.5 ]
'CAT'   [52.5, 51.2]
```

zip() function

- Iterate on multiple sequences in parallel

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

for colname, val in zip(columns, values):
    # Loops with colname='name'   val='GOOG'
    #            colname='shares' val=100
    #            colname='price'  val=490.1
    ...
```

- Common use: Making dictionaries

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

- Caution: Truncates to shortest input length

```
zip(['a','b','c'], [1,2]) ⟶ ('a', 1), ('b', 2)
```

Keeping a Running Count

- enumerate(*sequence* [, *start*])

```
names = ['IBM', 'YHOO', 'CAT' ]
for n, name in enumerate(names):
   # Loops with n=0 name='IBM'
   #            n=1 name='YHOO'
   #            n=2 name='CAT'
```

- Example: Line number tracking on a file

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

Iterating on Integers

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

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

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

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

- Note: The ending value is never included
- Caution: range() is often a "code smell" for problems being solved the "hard way"

Chaining Iteration

- Consider this code:

```
s1 = [1, 2, 3, 4]
s2 = [5, 6, 7, 8, 9, 10, 11]

for x in s1:
    ...           # x = 1, 2, 3, 4

for x in s2:
    ...           # x = 5, 6, 7, 8, 9, 10, 11
```

- Better: itertools.chain()

```
from itertools import chain

for x in chain(s1, s2):
    ...           # x = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
```

Unpacking Iterables

- Consider these iterables

```
a = (1, 2, 3)
b = [4, 5]
```

- Making lists and tuples (Python 3.5+)

```
c = [ *a, *b ]     # c = [1, 2, 3, 4, 5]    (list)
d = ( *a, *b )     # d = (1, 2, 3, 4, 5)    (tuple)
```

- It's subtle, but maybe better than using +

```
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple
>>>
```

Unpacking Dictionaries

- Consider these dicts

```
a = { 'name': 'GOOG', 'shares': 100, 'price':490.1 }
b = { 'date': '6/11/2007', 'time': '9:45am' }
```

- Combining into a single dict (Python 3.5+)

```
c = { **a, **b }

>>> c
{ 'name': 'GOOG', 'shares':100, 'price': 490.1,
  'date': '6/11/2007,'time': '9:45am' }
>>>
```

Argument Passing

- Iterables can be expanded in function calls

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

func(*a, *b)    #  func(1,2,3,4,5)
```

- Dictionaries can expand to keyword args

```
c = {'x': 1, 'y': 2 }

func(**c)   func(x=1, y=2)
```

- Combinations fine as long as positional go first

```
func(*a, **c)
func(*a, *b, **c)
func(0, *a, *b, 6, spam=37, **c)
```

Sequence Reductions

- sum(s), min(s), max(s)

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

- Boolean tests: any(s), all(s)

```
>>> s = [False, True, True, False]
>>> any(s)
True
>>> all(s)
False
>>>
```

Commentary

- Iteration is an essential Python skill
- It's also a bit of an art-form
- Can save a lot of time by knowing builtin functions (enumerate, zip, etc.)
- Also: The itertools standard library

Exercise 2.4

Time : 15 minutes

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 a sequence (and getting a copy)

```
result = sorted(s)
```

- It seems "simple" enough...

List Sorting

- Sort this list of dicts

```
portfolio = [
  {'name': 'AA', 'price': 32.2, 'shares': 100},
  {'name': 'IBM', 'price': 91.1, 'shares': 50},
  {'name': 'CAT', 'price': 83.44, 'shares': 150},
  {'name': 'MSFT', 'price': 51.23, 'shares': 200},
  {'name': 'GE', 'price': 40.37, 'shares': 95},
  {'name': 'MSFT', 'price': 65.1, 'shares': 50},
  {'name': 'IBM', 'price': 70.44, 'shares': 100}
]
```

- Question: How?
- By what criteria? (name, shares, price)

List Sorting

- You can control it using a "key function"

```
def stock_name(s):
    return s['name']

portfolio.sort(key=stock_name)
```

- Value returned by key func determines result

```
[{'name': 'AA', 'price': 32.2, 'shares': 100},
 {'name': 'CAT', 'price': 83.44, 'shares': 150},
 {'name': 'GE', 'price': 40.37, 'shares': 95},
 {'name': 'IBM', 'price': 91.1, 'shares': 50},
 {'name': 'IBM', 'price': 70.44, 'shares': 100},
 {'name': 'MSFT', 'price': 51.23, 'shares': 200},
 {'name': 'MSFT', 'price': 65.1, 'shares': 50}]
```

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

```
portfolio.sort(key=lambda s: s['name'])
```

- Creates an unnamed function that evaluates a single expression
- The above code is a shorter version of this

```
# Same as
def stock_name(s):
    return s['name']

portfolio.sort(key=stock_name)
```

Using lambda

- lambda is highly restricted
- Only a single expression is allowed and you can't use statements such as if, while, for, etc.
- Most common use is with sort()
- Sometimes seen with min(), max(), and other data handling functions

Exercise 2.5

Time : 10 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 = ['IBM', 'YHOO', 'CAT']
>>> a = [name.lower() for name in names]
>>> a
['ibm', 'yhoo', 'cat']
>>>
```

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: lines containing a substring

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

List Comprehensions

- General syntax

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

- What it means

```
result = []
for name 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 portfolio]
```

- Performing database-like queries

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

- Quick mathematics over sequences

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

Set/Dict Comprehensions

- List comprehension

```
>>> [ s['name'] for s in portfolio ]
[ 'AA', 'IBM', 'CAT', 'MSFT', 'GE', 'MSFT', 'IBM ' ]
>>>
```

- Set comprehension (eliminate duplicates)

```
>>> { s['name'] for s in portfolio }
{ 'GE', 'IBM', 'CAT', 'AA', 'MSFT' }
>>>
```

- Dict comprehension (make key/value pairs)

```
>>> { s['name']: 0 for s in portfolio }
{ 'GE': 0, 'IBM': 0, 'CAT': 0, 'AA': 0, 'MSFT': 0 }
>>>
```

Exercise 2.6

Time : 15 Minutes

Generator Expressions

- A variant of a list comprehension that produces the results incrementally
- Just slightly different syntax (parentheses)

```
nums = [1,2,3,4]
squares = (x*x for x in nums)
```

- To get the results, you use a for-loop

```
for n in squares:
    ...
```

Generators

- Unlike a list, a generators can only be used once (afterwards, they're useless)
- Example:

```
>>> nums = [1,2,3,4]
>>> squares = (x*x for x in nums)
>>> for n in squares:
        print(n, end=' ')

1 4 9 16
>>> for n in squares:
        print(n, end=' ')
        ←————————— notice no output (spent)
>>>
```

Using Generators

- Generators are useful in contexts where the result is only going to be used once and thrown away
- For example:

```
def sumsquares(nums):
    squares = (x*x for x in nums)
    total = sum(squares)
    return total
```

- Observe: squares is just some temporary value--no need to make a full list

Generator Arguments

- Generators expressions are sometimes embedded as the argument of functions that consume the values in a sequence

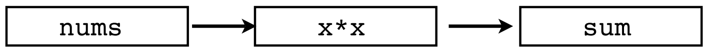
```
sum(x*x for x in nums)

print(','.join(str(x) for x in items))

if any(name.endswith('.py') for name in filenames):
     ...
```

- It looks funny at first, but this defines a generator expression which is passed as the input sequence to the function

Generator Arguments

- When embedded as a function argument, it acts as a kind of data filter/transform

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



- Critical point : Data is processed one item at a time--the extra step doesn't create a temporary list

Generator Functions

- A function that feeds iteration

```
def squares(nums):
    for x in nums:
        yield x*x      # Emit a value
```

- To get the results, you use a for-loop

```
for n in squares:
    ...
```

- This a more general form that can be used if the iteration processing is more complicated

Exercise 2.7

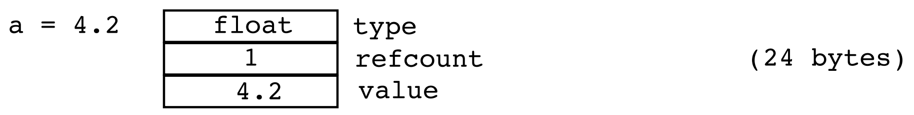
Time : 15 Minutes

Secrets of the Builtins

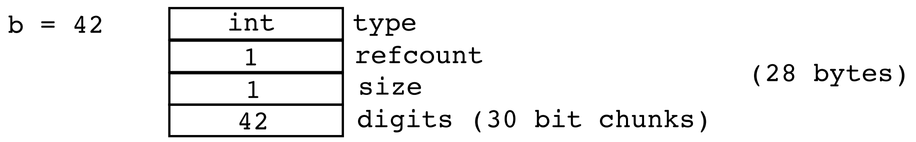
- Programmers use the built-in types without giving them much thought
- However, they have some subtle behavior that is worth knowing about
- Especially if you want to write better code

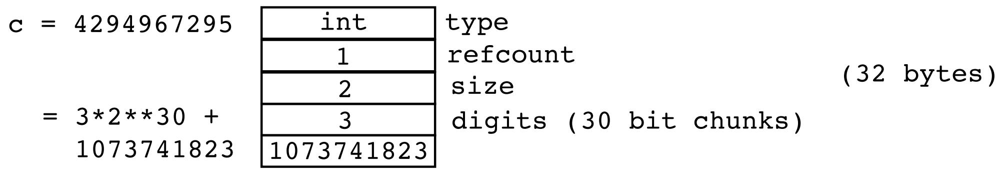
Object Representation

- Objects are often larger than you think



- May vary in size (depending on contents)





Object Sizes

- Use sys.getsizeof(obj) to investigate
- Example:

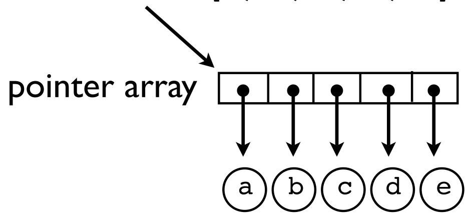
```
>>> import sys
>>> sys.getsizeof(4.2)
24
>>> sys.getsizeof(42)
28
>>> sys.getsizeof('hello world')
60
>>>
```

- This partly explains the large memory use in the first exercise (there's a lot of extra overhead)

Container Representation

- Container objects only hold references (pointers) to their stored values

```
items = [a, b, c, d, e]        # A list
```



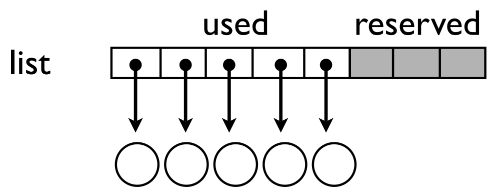
- All operations involving the container internals only manipulate the pointers (not the objects)

Memory Requirements

- Minimal storage requirements (64-bit)
  - Tuple : 48 bytes + 8 bytes/object
  - List : 64 bytes + 8 bytes/object
  - Set : 224 bytes + 16 bytes/object
  - Dict: 288 bytes + 24 bytes/object
  - Instance: 64 bytes + size of dict
- Note: Minimal space for sets/dicts already allow up to 5 objects to be stored

Over-allocation

- All mutable containers (lists, dicts, sets) tend to over-allocate memory so that there are always some free slots available

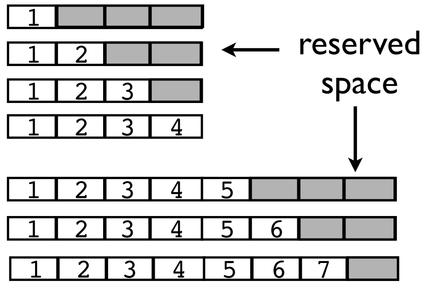


- This is a performance optimization
- Goal is to make appends, insertions fast

Example : List Memory

- Example of list memory allocation

```
items = []
items.append(1)
items.append(2)
items.append(3)
items.append(4)

items.append(5)
items.append(6)
items.append(7)
```

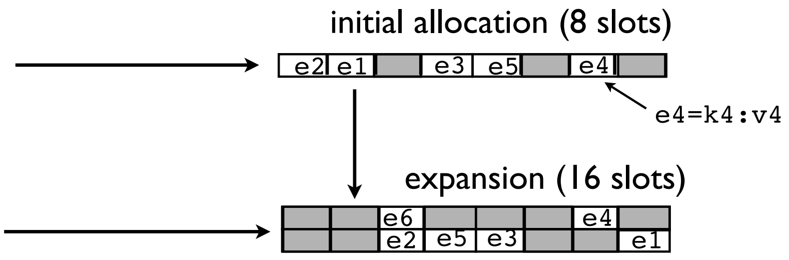


- Extra space means that most append() operations are very fast (space is already available, no memory allocation required)

Example : Dict Memory

- Example of dict memory allocation

```
d = {}
d[k1] = v1
d[k2] = v2
d[k3] = v3
d[k4] = v4
d[k5] = v5

d[k6] = v6
```



- On Python 2.X, growth is factor 4

Container Growth

- Memory use of containers grows in proportion to the number of stored values
- Lists : Space increases by ~12.5% when full
- Sets : Space increases by factor 4 when 2/3 full
- Dicts : Space increases by a factor 2 when 2/3 full (factor 4 on Python 2.X)

Memory and Instances

- Growth of dictionaries is of particular interest if you are using a lot of classes
- Each instance gets its own dictionary

```
0 - 5  attributes  (288 bytes)
6 - 10 attributes  (480 bytes)
```

- Fairly big jump in memory requirements once you go above 5 instance attributes
- Note: This is why you might use __slots__ or namedtuple as an alternative

Set/Dict Hashing

- Sets and dictionaries are based on hashing
- Keys are used to determine an integer "hashing value" (__hash__() method)

```
a = 'Python'
b = 'Guido'
c = 'Dave'

>>> a.__hash__()
-539294296
>>> b.__hash__()
1034194775
>>> c.__hash__()
2135385778
```

- Value used internally (implementation detail)

Key Restrictions

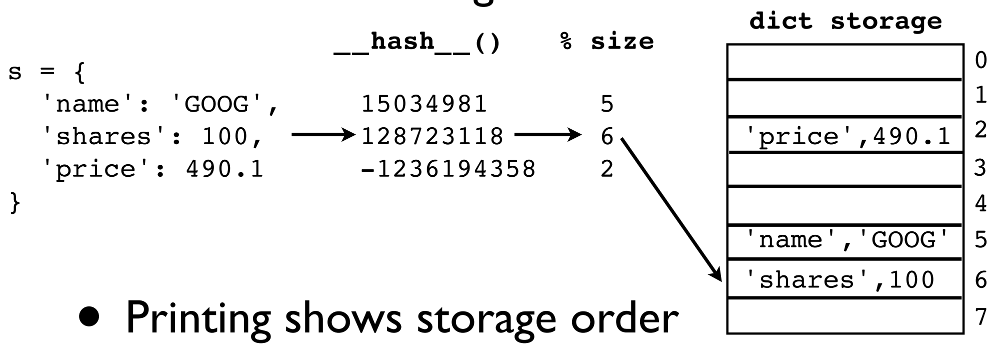
- Sets/dict keys restricted to “hashable” objects

```
>>> a = {'IBM','AA','AAPL'}
>>> b = {[1,2],[3,4]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

- This usually means you can only use strings, numbers, or tuples (no lists, dicts, sets, etc.)

Dict Storage Order

- Here's how hashing works:



```
                           __hash__()    % size
s = {
   'name': 'GOOG',          15034981        5
   'shares': 100,          128723118        6
   'price': 490.1          -1236194358      2
}
```

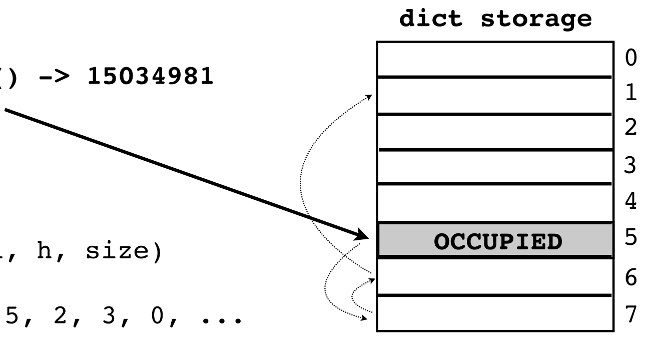
- Printing shows storage order

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

Collision Resolution

- Index is perturbed until an open slot found

```
key='name'
h = key.__hash__() -> 15034981
i = h % size -> 5
```



- Recurrence

```
i, h = perturb(i, h, size)

i = 7, 6, 1, 4, 5, 2, 3, 0, ...
```

- Every slot is tried eventually
- Works better if many open slots available

Exercise 2.8

Time : 10 Minutes

Final details on objects

- Variable assignment
- Copying
- Type checking

Understanding 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
```

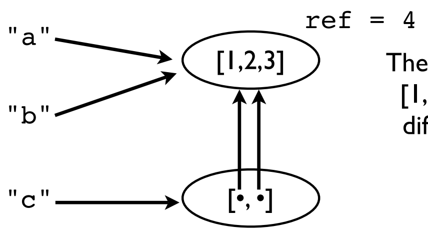
- A caution : assignment operations never make a copy of the value being assigned
- All assignments are merely reference copies (or pointer copies if you prefer)

Assignment Example

- Consider this code fragment:

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

- A picture of the underlying memory



There is only one list object [1,2,3], but there are four different references to it.

Assignment Caution

- 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]]
>>>
```

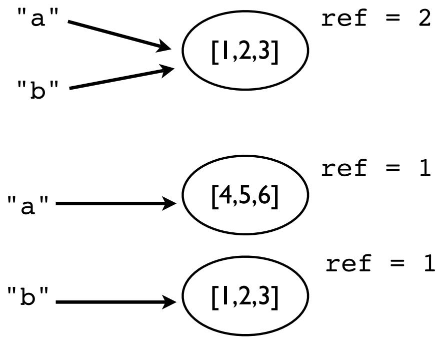
- Notice how a change to the original list shows up everywhere else (yikes!)
- This is because no copies were ever made-- everything is pointing at the same thing

Reassigning Values

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

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

```
a = [4,5,6]
```



- Key point: the name points to a different object

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
>>>
```

- Every object also has an integer identifier

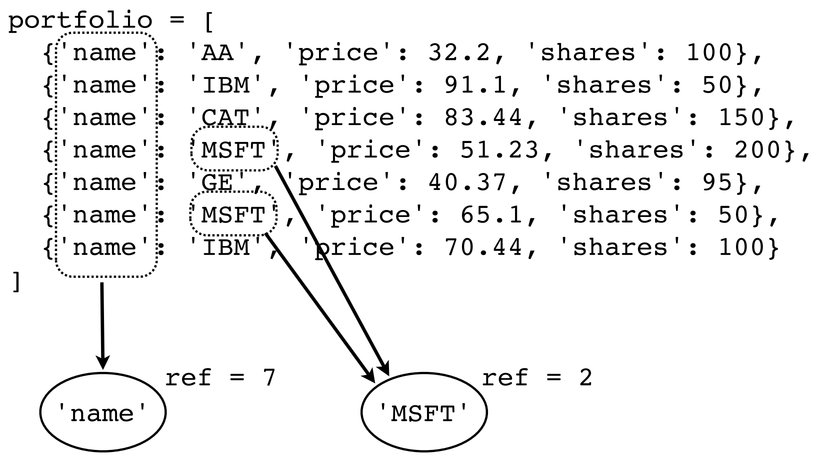
```
>>> id(a)
2774760
>>> id(b)
2774760
>>>
```

The object identifier is kind of like a pointer. If two names have the same id value, they're referring to the same object.

Exploiting Immutability

- Immutable values can be safely shared

```
portfolio = [
  {'name': 'AA', 'price': 32.2, 'shares': 100},
  {'name': 'IBM', 'price': 91.1, 'shares': 50},
  {'name': 'CAT', 'price': 83.44, 'shares': 150},
  {'name': 'MSFT', 'price': 51.23, 'shares': 200},
  {'name': 'GE', 'price': 40.37, 'shares': 95},
  {'name': 'MSFT', 'price': 65.1, 'shares': 50},
  {'name': 'IBM', 'price': 70.44, 'shares': 100}
]
```



- Sharing can save significant memory

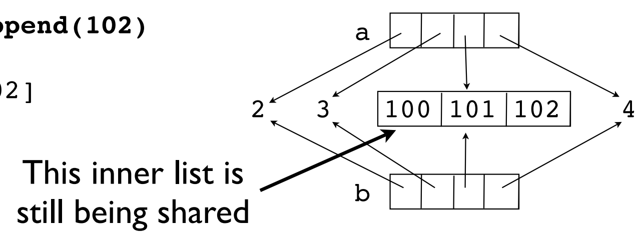
Shallow Copies

- Containers have methods for copying

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

- However, items are copied by reference

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



- Known as a "shallow copy"

Deep Copying

- Sometimes you need to makes 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)
<class 'int'>
>>> type(b)
<class 'str'>
```

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

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

Type Checking

- How to tell if an object is a specific type

```
if isinstance(a,list):    # Check if list
    print('a is a list')
```

- Checking for one of many types

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

- Advice: Don't go overboard with checking

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

Example: Emulating Cases

A big conditional with many cases

```
if op == '+':
    r = add(x, y)
elif op == '-':
    r = sub(x, y):
elif op == '*':
    r = mul(x, y):
elif op == '/':
    r = div(x, y):
```

➔

Reformulation using a dict of functions

```
ops = {
   '+' : add,
   '-' : sub,
   '*' : mul,
   '/' : div
}

r = ops[op](x,y)
```

- Key idea: Can make data structures from anything

Final Words

- Knowing how to effectively utilize Python's built-in types is an essential building block of writing solid (and efficient) code
- Considerable effort has gone into making Python's built-in types efficient
- You are unlikely to create a better solution

More Information

- A great source of information and recipes is the Python Cookbook (both the O'Reilly book and online)

http://code.activestate.com/recipes/langs/python/

- Try to find just about any Python paper or presentation by Raymond Hettinger (Python data structure expert extraordinaire)

Exercise 2.9

Time : 15 Minutes

Section 3

Classes and Objects

When to use Objects?

- Object oriented programming is largely concerned with the modeling of "behavior."
- An "object" consists of some internal state, but more importantly, has methods that make it do various things.
- The methods give an object its personality

An Example

- Data

```
host = ('www.python.org', 80)
```

- Behavior

```
c = Connection('www.python.org',80)
c.open()
c.send(data)
c.recv()
c.close()
```

- Data and behavior are bound together

The class statement

- Use 'class' to define a new object

```
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?
- Mostly, it's a set of functions that carry out various operations on so-called "instances"

Instances

- Instances are the actual "objects" that you manipulate in your program
- Created by calling the class as a function

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

- Emphasize: The class statement is just the definition (it does nothing by itself)

Instance Data

- Each instance has its own local data

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

- This data is initialized by __init__()

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

Any value stored on "self" is instance data

- There are no restrictions on the total number or type of attributes stored

Instance 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."

Attributes

- A word on terminology
- "Attribute" is anything accessed via (.)

```
>>> c.radius             # Attribute of an instance
4.0

>>> Circle.area          # Attribute of a class
<function Circle.area at 0x10e7f6400>

>>> import math
>>> math.pi              # Attribute of a module
```

- Don't read too much into it

History Lesson

- Python classes were one of the last major features implemented in the language
- Design goals included no changes in syntax (other than the class statement itself) and no changes to function scoping rules
- Hence : Instance methods are normal function definitions that simply receive the instance as the first argument (self)
- That's it

Odd Class Scoping

- Caution: Classes do not define a scope

```
def bar():
    print 'bar'

class Foo(object):
    def bar(self):
        print 'Foo.bar'
    def spam(self):
        bar()          # Calls global function bar()
        self.bar()     # Method bar() of self
```

- If want to operate on an instance, you always have to refer to it explicitly (e.g., self)

Exercise 3.1

Time : 10 Minutes

Manipulating Instances

- There are only three operations on instances

```
obj.attr              # Get an attribute
obj.attr = value      # Set an attribute
del obj.attr          # Delete an attribute
```

- Attributes can be freely added and deleted after an instance is created

```
>>> c = Circle(2.0)
>>> c.radius
2.0
>>> c.color = 'red'       # Add an attribute
>>> del c.radius          # Delete an attribute
>>>
```

Attribute Access Functions

- 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: Output

```
attributes = [ 'name', 'shares', 'price']
for attr in attributes:
    print(attr, '=', getattr(obj, attr))
```

- Note: getattr() has a useful default value arg

```
x = getattr(obj, 'x', None)
```

Method Invocation

- Invoking a method is a two-step process
- Lookup: The . operator
- Method call: The () operator

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

>>> c = Circle(2.0)
>>> a = c.area <------------------- Lookup
>>> a
<bound method Circle.area of <Circle object at 0x590d0>>
>>> a()
12.566370614359172 <------ 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

```
>>> c = Circle(2.0)
>>> c
<Circle object at 0x590d0>
>>> a = c.area
>>> a
<bound method Circle.area of <Circle object at 0x590d0>>
>>> a()
12.566370614359172
>>>
```

binding

Bound Methods

- Why would you care?
- Often a source of careless non-obvious errors

```
>>> c = Circle(2.0)
>>> print('Area : %0.2f' % c.area)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a float is required
>>>
```

Note missing ()

- Or devious behavior that's hard to debug

```
f = open(filename, 'w')
...
f.close
```

Oops. Didn't do anything at all

Bound Methods

- Under the covers, bound methods combine an instance (the "self") with a method

```
>>> c = Circle(2.0)
>>> a = c.area
>>> a
<bound method Circle.area of <Circle object at 0x590d0>>
>>> a.__self__
<Circle object at 0x590d0>
>>> a.__func__
<function area at 0x37cc30>
>>> a.__func__(a.__self__)
12.566370614359172
>>>
```

- Ponder it a bit, will return to it later

Exercise 3.2

Time : 15 Minutes

More on Class Definitions

- A class contains definitions that are shared by all instances of the class

```
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
```

- Shared : Defined once, used by all instances
- Example : There is one area() function that gets used by all instances created

Class Variables

- Classes may also define variables
- Known as "class variables"

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

- There are two access routes

```
>>> Circle.color               (On the class itself)
'black'
>>> c = Circle(2.0)
>>> c.color                    (On an instance of the class)
'black'
>>>
```

Using Class Variables

- Often used for settings applied to all instances

```
class Date(object):
   datefmt = '{year}-{month}-{day}'
   def __init__(self, year, month, day):
       self.year = year
       self.month = month
       self.day = day
   def output(self):
       print(self.datefmt.format(year=self.year,
                                 month=self.month,
                                 day=self.day))
```

- Possibly changed via inheritance

```
class USDate(Date):
   datefmt = '{month}/{day}/{year}'
```

Class Methods

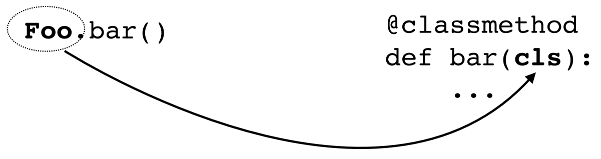
- A method that operates on the class itself

```
class Foo(object):
    @classmethod
    def bar(cls):
        print('Foo.bar', cls)
```

- It's invoked on the class, not an instance

```
>>> Foo.bar()
Foo.bar, <class '__main__.Foo'>
>>>
```

- The class is passed as the first argument



```
Foo.bar()                @classmethod
                         def bar(cls):
                             ...
```

Using Class Methods

- Class methods are often used as a tool for defining alternate initializers

```
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)
```

Notice how the class passed as an argument.

```
d = Date.today()
```

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)

Static Methods

- A function that's defined as part of a class, but does not operate on instances or the class

```
class Foo(object):
    @staticmethod
    def bar():
        print('Foo.bar')
```

- Example:

```
>>> Foo.bar()
Foo.bar
>>>
```

- Notice: There is no hidden self argument

Using Static Methods

- Uses vary:
  - Utility functions used by various methods
  - Instance management/tracking
  - Finalization, resource management
  - Certain design patterns
- Might improve code clarity--grouping related functionality together within a class

Exercise 3.3

Time : 10 Minutes

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

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
- That said, double underscores offer some benefits if using a lot of inheritance (attributes not visible in subclasses)
- Your 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      ───────→   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
>>>
```

- <u>No changes</u> 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
4
>>> c.area
50.26548245743669
>>> c.perimeter
25.132741228718345
```

Instance Variable

Computed Properties

- Commentary : Notice how there is no obvious difference between the attributes as seen by the user of the object

__slots__ Attribute

- You can restrict the set of attribute names

```
class Point(object):
   __slots__ = ('x', 'y')
   ...
```

- Produces errors for other attributes

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 20
>>> p.z = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Point' object has no attribute 'z'
```

- This is a performance optimization (uses less memory, runs faster)

__slots__ Cautions

- slots should only be used sparingly
- Be aware that it's presence can cause strange interaction with other parts of Python that are related to objects
- Advice : Do <u>not</u> use it except with classes that are simply going to serve as simple data structures

Commentary

- The features described so far cover virtually everything that you will see in most Python class definitions
- Essential pieces
  - Instance data (assignment in __init__)
  - Methods (instance, static, class)
  - Properties
  - Private attributes, __slots__

Exercise 3.4

Time : 15 Minutes

Inheritance

- A tool for specializing existing 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
- In a nutshell: Extending existing code

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 π 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()

"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 (even if you don't specify it in Python 3)
- Note: There is some historical baggage with Python 2. Inheriting from object is required to get a "new-style" class

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().spam()
        ...
```

notice how both methods have the same name.

- Use super() to do it
- Caution: Python 2 is different

```
r = super(Bar, self).spam()
```

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().__init__()    # init base
        self.radius = radius
```

- Again, you should use super() as shown

Multiple Inheritance

- You can specify 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 really tricky details (later)
- Don't do it unless you understand it

Using Inheritance

- Inheritance is often used as a code customization/extensibility feature
- For example, certain parts of a framework might involve inheriting from an existing class and redefining a handful of methods
- Idea: you add bits and pieces to existing code to make it do custom processing

Exercise 3.5

Time : 20 Minutes

Special Methods

- Classes can customize almost every aspect of their behavior
- This is done through special methods

```
class Point(object):
    def __init__(self, x, y):
        ...
    def __str__(self):
        ...
```

- There are dozens of these methods
- Instead of showing every possible customization, will show essential ones

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.

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)
a in x           x.__contains__(a)
```

- Definition in a class

```
class Container(object):
    def __len__(self):
        ...
    def __getitem__(self,a):
        ...
    def __setitem__(self,a,v):
        ...
    def __delitem__(self,a):
        ...
    def __contains__(self,a):
        ...
```

Methods: Mathematics

- Mathematical operators

```
a + b                  a.__add__(b)
a - b                  a.__sub__(b)
a * b                  a.__mul__(b)
a / b                  a.__div__(b)
a // b                 a.__floordiv__(b)
a % b                  a.__mod__(b)
a << b                 a.__lshift__(b)
a >> b                 a.__rshift__(b)
a & b                  a.__and__(b)
a | b                  a.__or__(b)
a ^ b                  a.__xor__(b)
a ** b                 a.__pow__(b)
-a                     a.__neg__()
~a                     a.__invert__()
abs(a)                 a.__abs__()
```

- Consult reference for further details

Instance Creation

- Instances are created in two steps

```
class Date(object):
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

d = Date(2012, 12, 21)
```

- Under the covers

```
d = Date.__new__(Date, 2012, 12, 21)
d.__init__(2012, 12, 21)
```

Using __new__

- Sometimes you might use __new__() directly

```
class Date(object):
    ...
    @classmethod
    def today(cls):
         t = time.localtime()
         self = cls.__new__(cls)
         self.year = t.tm_year
         self.month = t.tm_mon
         self.day = t.tm_mday
         return self

d = Date.today()
```

- Creates an instance, but bypasses __init__()

Defining __new__

- Classes may define __new__()

```
class A(object):
    @staticmethod
    def __new__(cls, x, y):
        ...
        return super().__new__(cls)
    def __init__(self, x, y):
        ...
```

- Not common, but sometimes used when altering some tricky aspect of instance creation
    - Instance caching
    - Immutability

__del__ method

- Classes might define a "destructor" method

```
class Connection(object):
    ...
    def __del__(self):
        # Cleanup statements
        ...
```

- Called when the reference count reaches 0
- Confusion: Not related to “del” operator

```
c = Connection()              # refcnt = 1
d = c                         # refcnt = 2

del d       # Doesn't call d.__del__()  (refcnt = 1)
c = None    # Calls c.__del__()  (refcnt = 0)
```

__del__ method

- Typical uses:
    - Proper shutdown of system resources (e.g., network connections)
    - Releasing locks (e.g., threading)
- Avoid defining it for any other purpose

Context Managers

- For resources, consider the use of the 'with' statement instead of relying on __del__()

```
with obj as val:  ---------->  val = obj.__enter__()
    statements
    statements
    statements
    ...
    statements
          ---------------->  obj.__exit__(ty, val, tb)
```

- Allows you to customize entry/exit steps

Context Managers

- Example:

```
class Manager(object):
    def __enter__(self):
        print('Entering')
        return self
    def __exit__(self, type, val, tb):
        print('Leaving')
        if type:
            print('An exception occurred')
```

- Example use:

```
>>> m = Manager()
>>> with m:
...      print('Hello World')
...
Entering
Hello World
Leaving
>>>
```

Exercise 3.6

Time : 15 Minutes

Code Reuse

- A major theme of object oriented programming concerns code reuse and making things extensible
- A big topic
- There are a number of common techniques

Interfaces

- Classes often serve as a kind of design specification or programming interface

```
class IStream(object):
    def read(self, maxbytes=None):
        raise NotImplementedError()
    def write(self, data):
        raise NotImplementedError()
```

- This class isn't used directly, but is usually included as a base class for other objects

```
class UnixPipe(IStream):
    def read(self, maxbytes=None):
        ...
    def write(self, data):
        ...
```

Abstract Base Classes

- Consider defining interfaces as an abstract base class (ABC) instead

```
from abc import ABC, abstractmethod

class IStream(ABC):
    @abstractmethod
    def read(self, maxbytes=None):
        pass
    @abstractmethod
    def write(self, data):
        pass
```

- Doesn't allow instantiation unless all of the abstract methods have been fully implemented

Abstract Base Classes

- ABCs may simplify type checking

```
def write_data(data, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected a Stream')
    ...
```

- ABCs catch careless usage errors

```
class UnixPipe(IStrem):
    def recv(self, maxbytes=None):
        ...
    def write(self, data):
        pass

>>> p = UnixPipe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class
UnixPipe with abstract methods read
>>>
```

Handler Classes

- Sometimes code will implement a general purpose algorithm, but will defer certain steps to a separately supplied handler object
- Sometimes known as the "strategy" design pattern.

Handler Classes

- Example :

Calls to handler methods

```
def print_table(records, fields, formatter):
    formatter.headings(fields)
    for r in records:
        rowdata = [getattr(r, fieldname, 'undef')
                   for fieldname in fields]
        formatter.row(rowdata)
```

- Notice how various steps of the algorithm are deferred to a separate handler object

Handler Classes

- Handlers have their own class definition

```
class TableFormatter(object):
    def headings(self, headings):
        raise NotImplementedError
    def row(self, rowdata):
        raise NotImplementedError
```

- The handler only contains the methods that need to be implemented/customized
- Important idea : Decoupling of the class that produces the table from the handler methods

Handler Classes

- Example Use

```
class TextTableFormatter(TableFormatter):
    def headings(self, headers):
        for h in headers:
            print('%10s' % h, end=' ')
        print()
        print(('-' * 10 + ' ') * len(headers))
    def row(self, rowdata):
        for d in rowdata:
            print('%10s' % d, end=' ')
        print()

formatter = TextTableFormatter(handler)
print_table(portfolio, ['name','shares'], formatter)
```

Commentary

- The use of handler classes is extremely common throughout the Python standard library (might be the most popular OO design pattern used in Python)
- Rationale : This approach provides flexibility
- Handlers are decoupled from implementation
- Allows handler code to be reused in other contexts (other classes can use the same handler objects).

Classes as a Template

- A class might implement a general-purpose algorithm, but delegate certain steps to a subclass
- Will illustrate with a simple example

Template Example

- A class that parses a CSV file into a list

```
class CSVParser(object):
    def parse(self, filename):
        with open(filename) as f:
             rows = csv.reader(f)
             self.headers = next(rows)
             records = []
             for row in rows:
                 record = self.make_record(row)
                 records.append(record)
        return records

    def make_record(self, row):
        raise RuntimeError('Must implement')
```

Step that must be implemented

- Note: Class is useless by itself

Template Example

- Using the template (use inheritance)

```
class DictCSVParser(DictParser):
    def make_record(self, row):
        return dict(zip(self.headers, row))

parser = DictCSVParser()
portfolio = parser.parse('portfolio.csv')
```

- Critical idea : User defines a small class that supplies the one missing piece, but most of the real functionality is in the base class

Exercise 3.7

Time : 15 Minutes

Advanced Inheritance

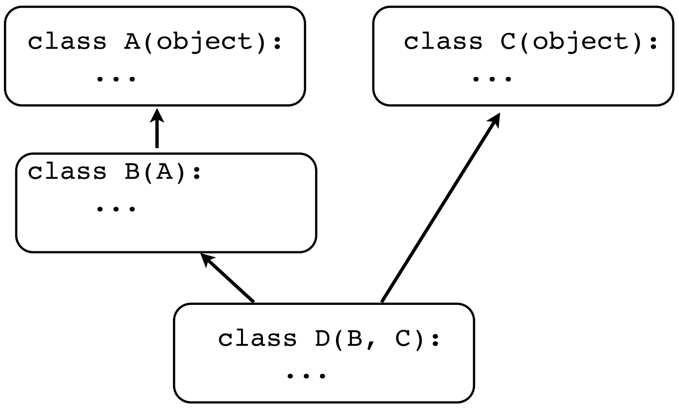
- Recall: Inheritance is a tool for code reuse (customization and extension)

```
class Parent(object):
    def spam(self):
        ...

class Child(Parent):
    def spam(self):
        print('Different spam')
        super().spam()
```

- Child classes can customize their parents
- Sometimes see use of super() function (shown)

Multiple Inheritance

- Classes can have multiple parents



```
class A(object):
    ...

class C(object):
    ...

class B(A):
    ...

class D(B, C):
    ...
```

- The child will inherit features from all parents
- But, it's a lot sneakier than this

Cooperative Inheritance

- Python uses "cooperative multiple inheritance"
- Big idea: A child class can specifically arrange its parents to cooperate with each other

```
class Child(Parent1, Parent2, Parent3):
    ...
```

- The order of the parents has significance
- Attribute search may jump parent-to-parent

```
class Child(Parent1, Parent2, Parent3):
    ...
```

Cooperative Inheritance

- Example: Consider this arrangement

```
class Parent(object):
    def spam(self):
        print('Parent')
```

```
class A(Parent):
    def spam(self):
        print('A')
        super().spam()
```

```
class B(Parent):
    def spam(self):
        print('B')
        super().spam()
```

- Now, this:

```
class Child(A,B):
    pass
```

```
>>> c = Child()
>>> c.spam()
A
B
Parent
>>>
```

it's gone sideways!

Cooperative Inheritance

- There are applications

- You can make collections of classes that are meant to be stacked together to make more interesting things

Mixin Classes

- A mixin is a class whose purpose is to add extra functionality to other class definitions
- Idea : If a user implements some basic features in their class, a mixin can be used to fill out the class with extra functionality
- Sometimes used as a technique for reducing the amount of code that must be written

Mixin Example

- Here's a class with no notable parent

```
class LoggedMixin(object):
    def __getitem__(self, key):
        print('getitem:', key)
        return super().__getitem__(key)
```

- It's useless by itself

```
>>> d = LoggedMixin()
>>> d['spam']
getitem: spam
Traceback (most recent call last):
  File "<stdin>", line 4, in __getitem__
AttributeError: 'super' object has no attribute '__getitem__'
>>>
```

- It has to be combined with another class

Mixin Example

- Example: mixing with a list

```
>>> class LogList(LoggedMixin, list):
        pass

>>> items = LogList(['a','b','c'])
>>> items[0]
getitem: 0
'a'
>>>
```

- Example: mixing with a Counter

```
>>> from collections import Counter
>>> class LogCounter(LoggedMixin, Counter):
        pass
>>> c = LogCounter()
>>> c['spam'] += 10
getitem: spam
>>> c['spam']
getitem: spam
10
```

Use of Mixins

- Mixin classes are sometimes used as a way to add optional features to more basic objects
- For example, added thread support, persistence, etc.
- User assembles an object from the different parts that they're going to use

Exercise 3.8

Time : 15 Minutes

Section 4

Inside Python Objects

Overview

- Inner details on how Python objects work
- Object representation
- Attribute binding
- Type checking
- Descriptors
- Attribute special methods

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 Objects

- User-defined objects 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
```

```
                          {
                            'name' : 'GOOG',
self.__dict__  ---------->  'shares' : 100,
                            'price' : 490.10
                          }
```

instance data

Dicts and Instances

- Critical point : Each instance gets its own private dictionary

```
                                      {
                                        'name' : 'GOOG',
s = Stock('GOOG',100,490.10) ------->   'shares' : 100,
t = Stock('AAPL',50,123.45)             'price' : 490.10
                          \           }
                           \
                            \-------> {
                                        'name' : 'AAPL',
                                        'shares' : 50,
                                        'price' : 123.45
                                      }
```

- So, if you created 100 instances of some class, there are 100 dictionaries sitting around holding data

Dicts and Classes

- A dictionary holds the members 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
```

```
                        {
                         'cost' : <function>,
Stock.__dict__ ------->  'sell' : <function>,
                         '__init__' : <function>,
                        }
```

methods

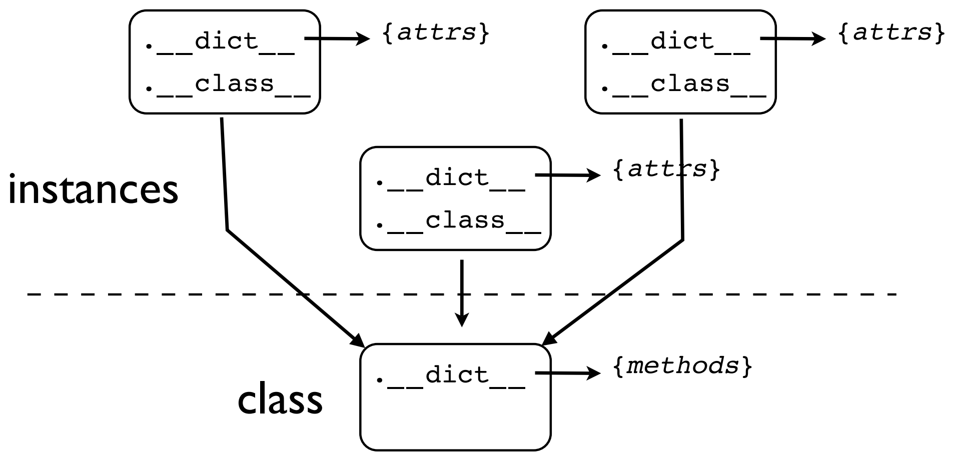
Instances and Classes

- 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 <u>all</u> 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'}
   >>>
```

Reading Attributes

- Suppose you read an attribute on an instance

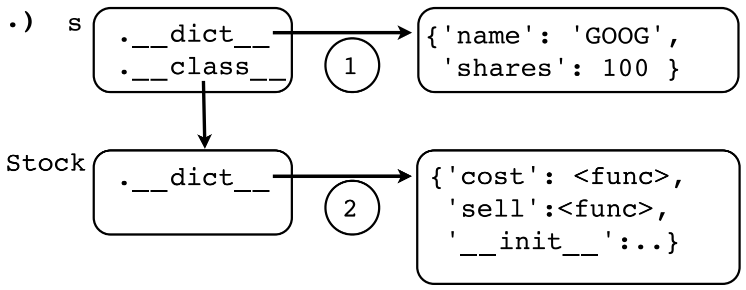
```
x = obj.name
```

- 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

```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```



- This lookup scheme is how the members of a class get shared by all instances

Exercise 4.1

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

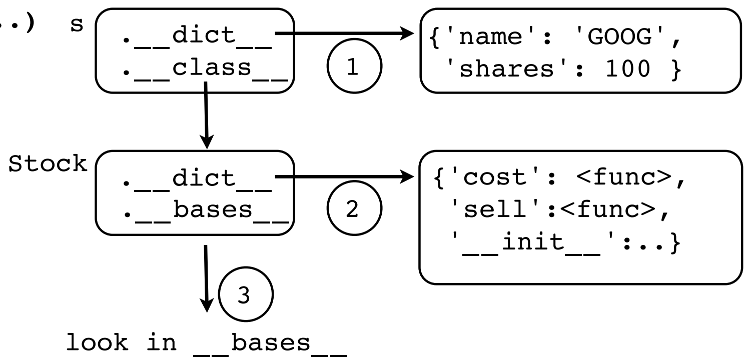
```
>>> A.__bases__
(<class '__main__.B'>,<class '__main__.C'>)
>>>
```

- This provides a link to parent classes
- This link simply extends the search process used to find attributes

Reading Attributes

- First check in local __dict__
- If not found, look in __dict__ of class
- If not found in class, look in base classes

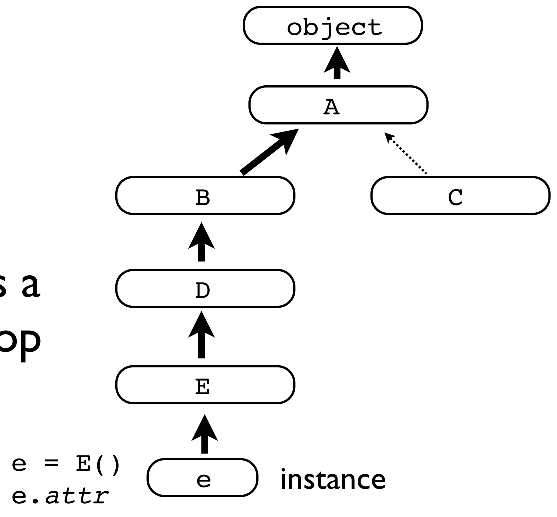
```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```



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

The MRO

- The inheritance chain is precomputed and stored in an "MRO" attribute on the class

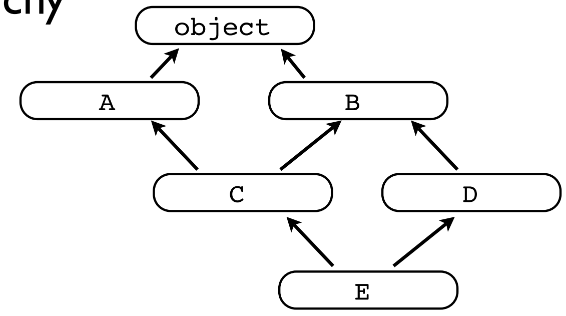
```
>>> E.__mro__
(<class '__main__.E'>,  <class '__main__.D'>,
 <class '__main__.B'>,  <class '__main__.A'>,
 <type 'object'>)
>>>
```

- "Method Resolution Order"
- To find attributes, Python walks the MRO
- First match wins

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
```

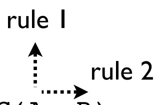
- A similar search process is carried out, but there is an added complication in that there may be many possible search paths

Multiple Inheritance

- Python uses "cooperative multiple inheritance"
- There are some ordering rules:

<u>Rule 1</u>: Children before parents
<u>Rule 2</u>: Parents go in order

- Inheritance works in two directions (up the hierarchy, across the list of parents)



```
class C(A, B):
    ...
```

Multiple Inheritance

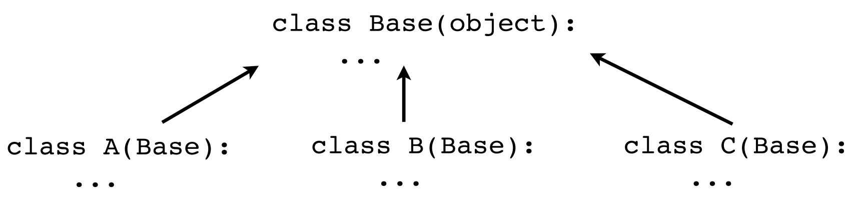
- Multiple inheritance hierarchy is flattened

```
>>> D.__mro__
(<class '__main__.D'>,  <class '__main__.B'>,
 <class '__main__.C'>,  <class '__main__.A'>,
 <type 'object'>)
>>>
```

- Calculated using the C3 Linearization algorithm
- A constrained merge sort of parent MROs
- An ordering based on "the rules"

Multiple Inheritance

- Consider classes with a common parent



```
                    class Base(object):
                        ...

class A(Base):       class B(Base):       class C(Base):
    ...                  ...                  ...
```

- All children of a common parent go first

```
class D(A,B,C):
    ...
```

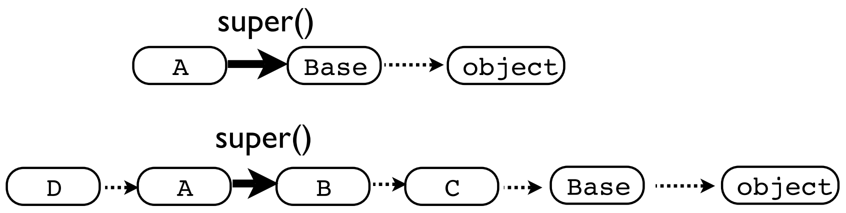
$$\begin{tikzcd} \text{MRO} & \boxed{D} \arrow[r, dotted] & \boxed{A} \arrow[r, dotted] & \boxed{B} \arrow[r, dotted] & \boxed{C} \arrow[r, dotted] & \boxed{Base} \end{tikzcd}$$

Why super()?

- Always use super() when overriding methods

```
class A(Base):
   def spam(self):
       ...
       return super().spam()
```

- super() delegates to the <u>next class</u> on the MRO



- Tricky bit: You don't know what it is

super() Explained

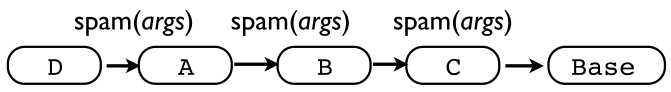
- super() is one of the most poorly understood Python features

```
class A(Base):                      class A(Base):
    def spam(self):        vs.          def spam(self):
        Base.spam(self)                     super().spam()
```

- These two classes are <u>not</u> the same
- super() binds to the <u>next implementation</u> that is defined according to the instance's MRO
- It's <u>not</u> necessarily the immediate parent

Designing for Inheritance

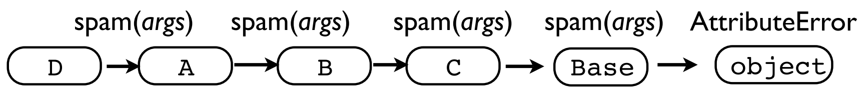
- Rule 1: Compatible Method Arguments



- Overridden methods must have a compatible signature across the entire hierarchy
- Remember: super() might not go to the immediate parent
- Tip: If there are varying method signatures, use keyword arguments

Designing for Inheritance

- Rule 2: Method chains must terminate



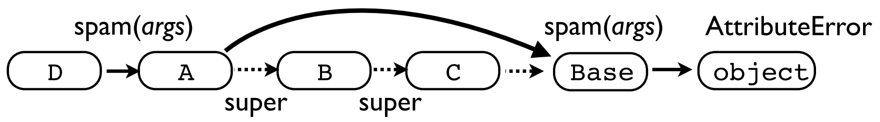
- You can't use super() forever--some class has to terminate the search chain

```
class Base(object):
    def spam(self):
        pass
```

- Typically the role of an abstract base class

Designing for Inheritance

- Rule 3: use super() everywhere



- Direct parent calls might explode heads

```
class A(Base):
    def spam(self):
        Base.spam(self)    # NO!
```

- If multiple inheritance is used, a direct parent call will probably violate the MRO

More Information

- "Python's super() considered super"

  http://rhettinger.wordpress.com/2011/05/26/super-considered-super/

- Associated PyCon Presentation

  http://pyvideo.org/video/3413/super-considered-super

Exercise 4.2

Time : 25 Minutes

Dicts and Classes (Reprise)

- Recall, a dictionary holds class members

```
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
```

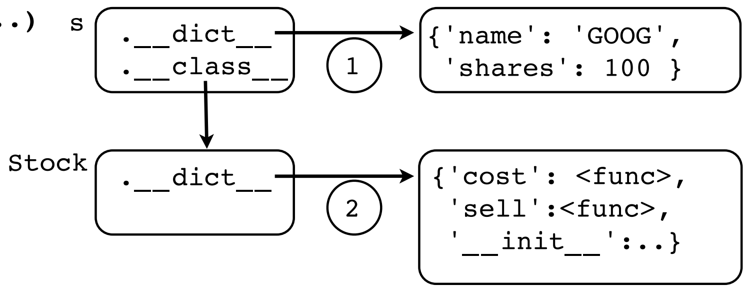
```
                          {
                           'cost' : <function>,
Stock.__dict__ ---------->  'sell' : <function>,
                           '__init__' : <function>,
                          }
```

methods

Reading Attributes (Reprise)

- Recall that a two-step process is used to locate attributes on objects

```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
>>>
```



- This is mostly correct
- Except for the extra hidden magic (not shown)

Attribute Binding

- Access to attributes of classes involves one extra processing step
- Something known as the "descriptor protocol"
- It's so sneaky that most Python programmers don't even know it exists
- Yet, it holds the whole object system together

Descriptor Protocol

- Whenever an attribute is accessed on a class, the attribute is checked to see if it is an object that looks like a so-called "descriptor"
- A *descriptor* is an object with one or more of the following special methods

```
d.__get__(obj, cls)         # Required
d.__set__(obj, value)       # Optional
d.__delete__(obj)           # Optional
```

- If a descriptor is detected, one of the above methods gets triggered on access

Descriptor Demo

- Here is a class that implements a dummy descriptor (with prints for debugging)

```
class Descriptor(object):
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls):
        print('%s:__get__' % self.name)
    def __set__(self, instance, value):
        print('%s:__set__ %s' % (self.name, value))
    def __delete__(self, instance):
        print('%s:__delete__' % self.name)
```

- Basically, a descriptor is just an object with get, set, and delete methods

Descriptor Demo

- Descriptors are placed in class definitions

```
class Foo(object):
    a = Descriptor('a')
    b = Descriptor('b')
```

- Now, watch what happens on access:

```
>>> f = Foo()
>>> f.a
a:__get__
>>> f.a = 42
a:__set__ 42
>>> del f.a
a:__delete__
>>> f.b
b:__get__
>>>
```

Descriptor Demo

- Descriptors are presented with information about the instance, class, and values

```
class Descriptor(object):
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls):
        print('%s:__get__' % self.name)
    def __set__(self, instance, value):
        print('%s:__set__ %s' % (self.name, value))
    def __delete__(self, instance):
        print('%s:__delete__' % self.name)
```

```
f = Foo()
f.a
f.a = 42
del f.a
```

- Confusion: self is the descriptor itself, instance is the object it's operating on.

Descriptor Storage

- Descriptors store and retrieve data

```
class Descriptor(object):
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls=None):
        return instance.__dict__[self.name]
    def __set__(self, instance, value):
        instance.__dict__[self.name] = value
```

Direct manipulation of the instance dictionary

- Example:

```
class Foo(object):
    a = Descriptor('a')
    b = Descriptor('b')

f = Foo()
f.a = 23     # Stores value in f.__dict__['a']
```

Descriptor Binding

- Descriptors always override __dict__

```
class Foo(object):
    a = Descriptor('a')
    b = Descriptor('b')
```

- Modify the instance dict and try accessing

```
>>> f = Foo()
>>> f.__dict__['a'] = 42
>>> f.__dict__
{'a': 42}
>>> f.a
a:__get__
>>>
```

notice how the descriptor runs regardless the value in the instance dictionary

Who Cares?

- Every major feature of classes is implemented using descriptors
    - Instance methods
    - Static methods (@staticmethod)
    - Class methods (@classmethod)
    - Properties (@property)
    - __slots__
- Descriptors provide the glue that connects instances and classes together in the runtime

Descriptors in Action

- Recall that . and () are separate operations

```
>>> s = Stock('GOOG',100,490.10)
>>> s.cost
<bound method Stock.cost of <__main__.Stock object at
0x37e250>>
>>> s.cost()
49010.0
>>>
```

- Focus on that "bound method" result
- How did that get created? Magic?
- No, a descriptor did that.

Descriptors in Action

- Behind the scenes of method lookup

class attribute lookup

descriptor check and invocation

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

>>> value = Stock.__dict__['cost']
>>> value
<function cost at 0x378770>
>>> hasattr(value,"__get__")
True
>>> result = value.__get__(s,Stock)
>>> result
<bound method Stock.cost of <__main__.Stock object at
0x37e250>>
>>> result()
49010.0
>>>
```

- Functions are descriptors where __get__() creates the bound method object

Descriptors and Properties

- Consider a class with a property attribute

```
class Foo(object):
    @property
    def name(self):
        return self._name
    @name.setter
    def name(self,value):
        self._name = value
```

- A property is also a descriptor

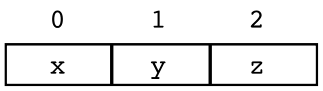
```
>>> f = Foo()
>>> p = Foo.__dict__['name']
>>> p
<property object at 0x3759c0>
>>> p.__set__(f,"Guido")    # Same as f.name='Guido'
>>> p.__get__(f,Foo)        # Same as f.name
'Guido'
>>> f.name
'Guido'
```

Descriptors and __slots__

- Consider a class with slots

```
class Foo(object):
    __slots__ = ('x','y','z')
    ...
```

- Internally, an array is allocated



- Each slot name is used to create a descriptor that simply gets or sets values in the appropriate array position (internals are implemented in C and hard to view though)

Descriptor Commentary

- Descriptors are one of Python's most powerful customizations (you own the dot)
- Experts can create their own custom descriptors and use them to change what happens in the low levels of the object system
- Often used in advanced programming frameworks and as an encapsulation tool

Descriptor Application

- A common use of descriptors is in describing data (e.g., Object Relational Mapping, etc.)

```
class Stock(object):
    name   = String('name',maxlen=8)
    shares = Integer('shares')
    price  = Real('price')
```

- Provide more precise control than properties.
- Results in less repetitive code

Descriptor Application

- Example descriptor code:

```
class Integer(object):
    def __init__(self, name):
        self.name = name
    def __get__(self, instance, cls):
        return instance.__dict__[self.name]
    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an integer')
        instance.__dict__[self.name] = value
```

- Minor note: __get__() could be omitted

Tricky Bits with __get__

- __get__ can be accessed in two ways

```
class Foo(object):
    a = Descriptor('a')
```

- Through an instance (bound)

```
f = Foo()
f.a
```

- On the class definition itself (unbound)

```
Foo.a
```

- Example : Bound vs. unbound methods

Tricky Bits with __get__

- Recommended __get__ implementation

```
class Descriptor(object):
    def __get__(self, instance, cls):
        if instance is None:
            # If no instance given, return the descriptor
            # object itself
            return self
        else:
            # Return the instance value
            return instance.__dict__[self.name]
```

- Always check for presence of an instance (instance). If None, return the descriptor itself

Method Descriptors

- A weaker descriptor that only has __get__

```
class MethodDescriptor(object):
    def __get__(self, instance, cls):
        print('Getting!')
```

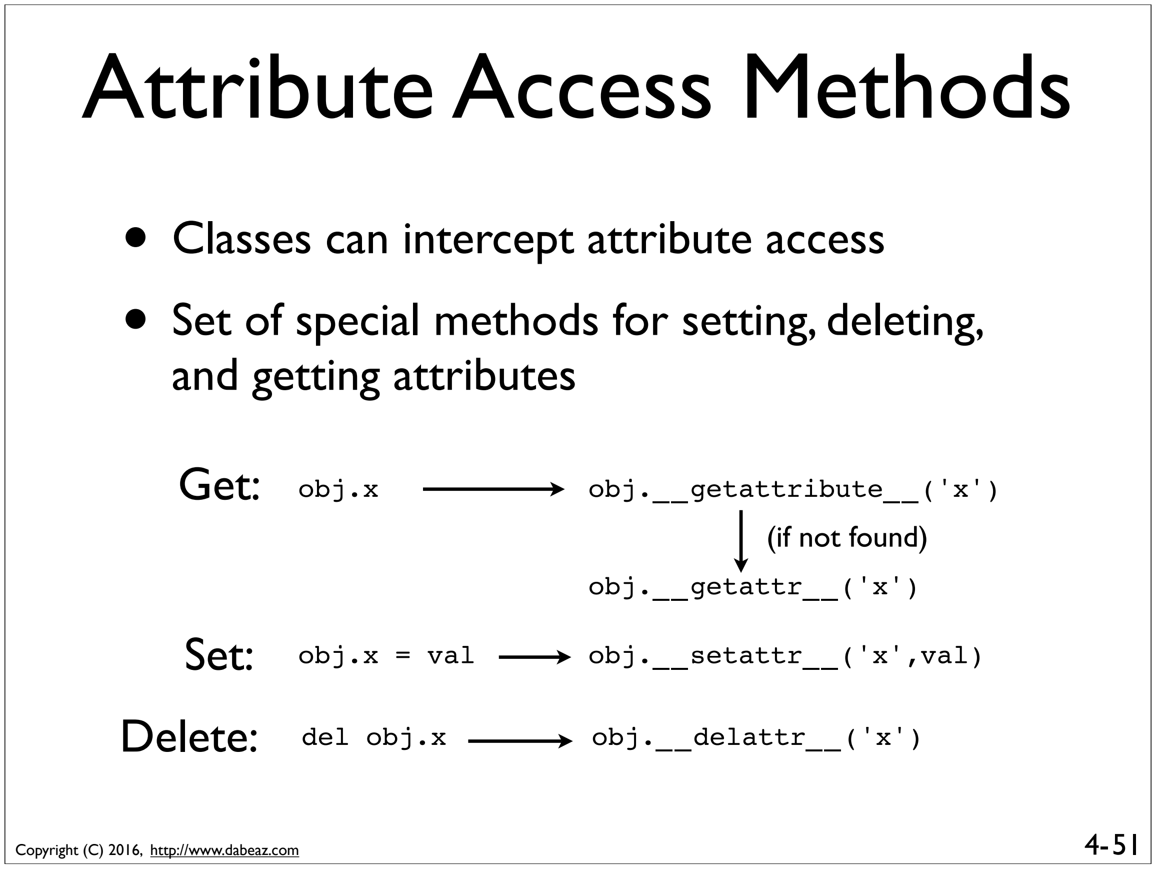
- Only triggered if obj.__dict__ doesn't match

```
class Foo(object):
    a = MethodDescriptor("a")

>>> f = Foo()
>>> f.a
Getting!
>>> f.__dict__['a'] = 42
>>> f.a
42
>>>
```

Notice how the value in the dictionary hides the descriptor

Exercise 4.3

Time : 15 Minutes



Attribute Access Methods

- Classes can intercept attribute access
- Set of special methods for setting, deleting, and getting attributes

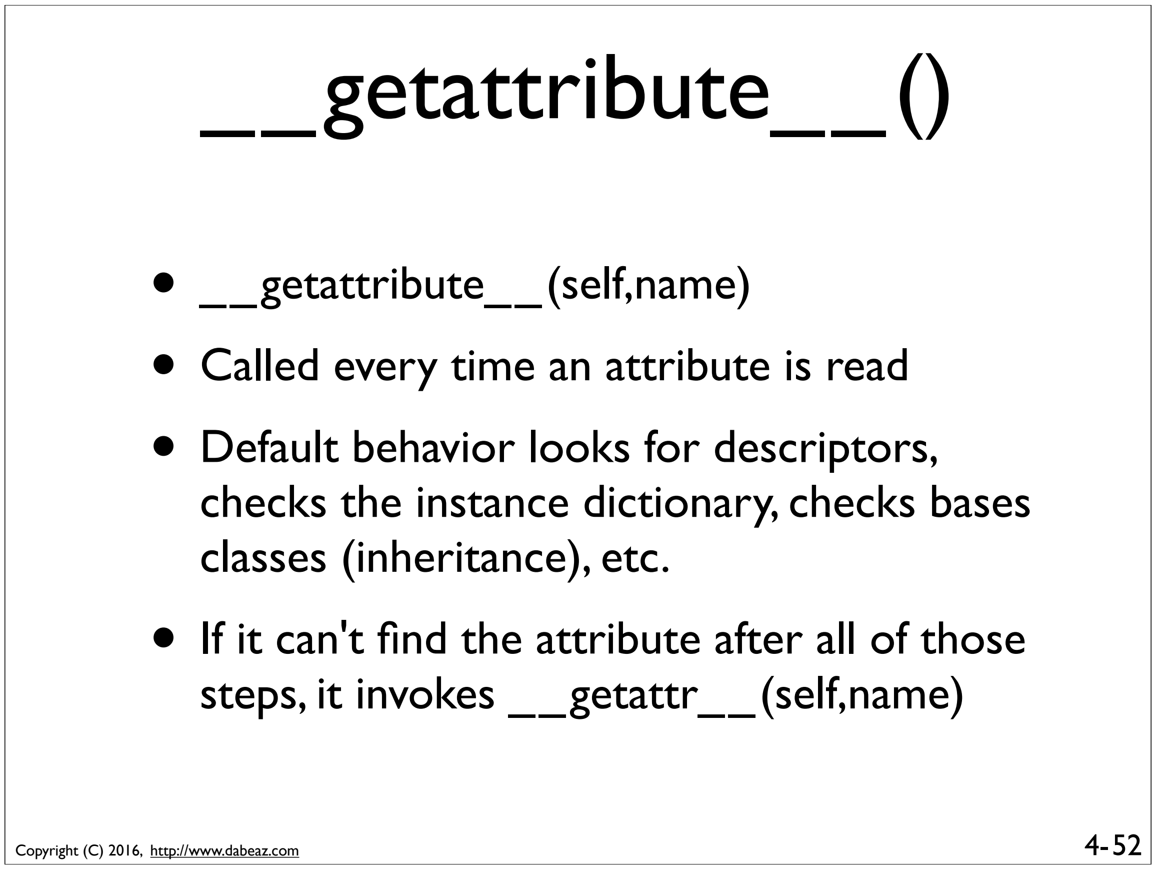
Get: `obj.x` ⟶ `obj.__getattribute__('x')`

↓ (if not found)

`obj.__getattr__('x')`

Set: `obj.x = val` ⟶ `obj.__setattr__('x',val)`

Delete: `del obj.x` ⟶ `obj.__delattr__('x')`



__getattribute__()

- __getattribute__(self,name)
- Called every time an attribute is read
- Default behavior looks for descriptors, checks the instance dictionary, checks bases classes (inheritance), etc.
- If it can't find the attribute after all of those steps, it invokes __getattr__(self,name)

__getattr__() method

- __getattr__(self,name)
- A failsafe method. Called if an attribute can't be found using the standard mechanism
- Default behavior is to raise AttributeError
- Sometimes customized

__setattr__() method

- __setattr__(self,name,value)
- Called every time an attribute is set
- Default behavior checks for descriptors, stores values in the instance dictionary, etc.

__delattr__() method

- __delattr__(self,name)
- Called every time an attribute is deleted
- Default behavior checks for descriptors and deletes from the instance dictionary

Customizing Access

- A class can redefine the attribute access methods to implement custom processing
- The most common application of this is for creating wrapper objects, proxies, and other similar kinds of objects

Example : Proxy

- Consider this class

```
class Proxy(object):
    def __init__(self,obj):
        self._obj = obj
    def __getattr__(self,name):
        print('getattr:', name)
        return getattr(self._obj, name)
```

- It holds an internal reference to an object
- Attribute access is redirected to held object

Example : Proxy

- Example use:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area()
50.26548245743669

>>> p = Proxy(c)
>>> p
<__main__.Proxy object at 0x37f130>
>>> p.radius
getattr: radius
4.0
>>> p.area()
getattr: area
50.26548245743669
>>>
```

Notice how attribute access gets captured by __getattr__ and then redirected to the original object

Example: Delegation

```
class A(object):
    def foo(self):
        print('A.foo')
    def bar(self):
        print('A.bar')

class B(object):
    def __init__(self):
        self._a = A()
    def bar(self):
        print('B.bar')
        self._a.bar()
    def __getattr__(self, name):
        return getattr(self._a, name)
```

- Example:

```
>>> b = B()
>>> b.foo()
A.foo
>>> b.bar()
B.bar
A.bar
>>>
```

- Sometimes used as an alternative to inheritance

Delegation Caution

- __getattr__ doesn't apply to special methods (e.g., __len__, __getitem__, etc.)
- Must delegate manually (if needed)

```
class B(object):
    def __init__(self):
        self._a = A()

    def __getitem__(self, index):
        return self._a[index]

    def __getattr__(self, name):
        return getattr(self._a, name)
```

Exercise 4.4

Time : 20 Minutes

Section 5

Being "Pythonic"

Overview

- Function design
- Error handling and Logging
- Object design
- Testing
- Optimization

Functions

- Functions are a basic building block
- Top-level functions in a module
- Methods of a class
- Almost all of your code will live in a function
- There is an official style guide: PEP 8

Naming Conventions

- Functions should use lowercase names and _

```
def read_data(filename):        def readData(filename):
    ...                             ...
```

Yes No

- Use a leading _ for internal/private funcs

```
def _internal_func():
    ...
```

Function Design

- Functions should <u>ideally operate</u> on their passed inputs and return a proper result

Yes

```
def read_data(filename):
    records = []
    f = open(filename)
    ...
    return records

result = read_data('data.csv')
```

No

```
filename = 'data.csv'
records = []

def read_data():
    f = open(filename)
    ...
    records.append(r)
```

- Avoid global variables
- Avoid hidden side-effects and magic

Side Effects/Mutability

- Don't return a result when mutating state

```
>>> names = ['Paula','Dave','Lewis','Thomas']
>>> names.sort()
>>> names
['Dave', 'Lewis', 'Paula', 'Thomas']
>>>
```

notice: no result is returned

- Contrast with string methods

```
>>> s = 'hello world'
>>> s.replace('hello', 'hello cruel')
'hello cruel world'
>>> s
'hello world'
>>>
```

- You want a clear indication of mutability

Argument Passing

- Arguments are passed as objects, never copied
- If you pass a mutable object (list, dict, etc.), changes affect the original value

```
def foo(items):
    items.append(4)

nums = [1, 2, 3]
foo(nums)
print(nums)   # [1, 2, 3, 4]
```

- Rule of thumb: Don't modify inputs

Optional Arguments

- Sometimes you want an optional argument

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

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

```
d = read_data('data.csv')
e = read_data('data.csv', debug=True)
```

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

Keyword Arguments

- Prefer keywords for passing optional arguments

```
a = read_data('data.csv', debug=True)    # YES!
b = read_data('data.csv', True)          # NO!
```

- Keywords result in better code clarity
- You can force the use of keyword arguments

```
def read_data(filename, *, debug=False):
    ...
```

All arguments after the * must be given as by keyword

Default Value Binding

- Caution: Default values get set <u>once</u> at the time of <u>function definition</u>

```
DEBUG=False
def read_data(filename, debug=DEBUG):
    print(filename, debug)

>>> read_data('data.csv')
data.csv False
>>> DEBUG=True
>>> read_data('data.csv')
data.csv False
>>>
```

- Once a function is defined, you can't change the default value

Default Values

- Don't use mutable values as defaults

```
def spam(a, items=[]):
    items.append(a)
    return items
```

- The default value is only created once for the whole program--mutations are "sticky"

```
>>> spam(1)
[1]
>>> spam(2)
[1, 2]
>>> spam(3)
[1, 2, 3]
```

- A really bad idea if the default value escapes the function (e.g., returned as a result)

Default Values

- Advice: Only use immutable values such as None, True, False, numbers, or strings

```
def spam(a, items=None):
    if items is None:
       items = []
    items.append(a)
    return items
```

- This avoids the problem of the default being modified by accident

```
>>> spam(1)
[1]
>>> spam(2)
[2]
>>>
```

Doc Strings

- Functions should have a doc string

```
def add(x, y):
    '''
    Adds x and y together.
    '''
    return x + y
```

- Feeds the help() command and development tools
- Important: there are no type signatures or other details to help people reading your code. The more information you provide, the better.

Type Hints (PEP 484)

- Optional annotations can indicate types

```
def add(x:int, y:int) -> int:
    '''
    Adds x and y together.
    '''
    return x + y
```

- The type hints do nothing, but may be useful for code checkers, documentation, IDEs, etc.

```
>>> help(add)
Help on function add in module __main__:

add(x:int, y:int) -> int
    Adds x and y
```

Assertions/Contracts

- Assertions are runtime checks

```
def add(x, y):
    '''
    Adds x and y
    '''
    assert isinstance(x, int)
    assert isinstance(y, int)
    return x + y
```

- Function will fail on bad input

```
>>> add(2, 3)
5
>>> add('2', '3')
Traceback (most recent call last):
...
AssertionError
>>>
```

Assertions/Contracts

- Assertions are not meant to check user inputs
- Validating program invariants (internal conditions that must always hold true)
- Failure indicates a programming error and assigns blame (e.g., to the caller)
- Can be disabled (python -O)

```
bash % python3 -O prog.py
```

Exercise 5.1

Time : 10 Minutes

Function Error Checking

- As you know, exceptions indicate errors

```
raise RuntimeError("You're dead")
```

- Exceptions can be caught

```
try:
    statements
    ...
except RuntimeError as e:
    # Handle the runtime error
    ...
```

- The mechanics of exception handling is usually straightforward, but proper usage is often a lot trickier than it looks

What Exceptions to Handle?

- Functions should only handle exceptions where recovery is possible (and makes sense):

```
def read_csv(filename):
    f = open(filename)
    for row in csv.reader(f):
        try:
            name = row[0]
            shares = int(row[1])
            price = float(row[2])
        except ValueError as e:
            print('Bad row:', row)
            continue
        ...
```

- Let all other exceptions propagate--they usually indicate a more serious problem

Example

- Don't worry about things like this

```
>>> read_csv('bogus.csv')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "reader.py", line 10, in read_csv
    f = open(filename)
FileNotFoundError: [Errno 2] No such file or directory:
'bogus.csv'
>>>
```

- There is no sensible recovery. The failure is someone else's problem.

Catching All Errors

- Never catch all exceptions unless you report/ record the actual exception that occurred

```
try:
    # Some complicated operation
    ...
except Exception as e:
    print("Sorry, it didn't work.")
    print("Reason:", e)
    ...
```

- Not reporting actual exception information is the fastest way to create undebuggable code

Ignoring Errors

- No! No! No!

```
try:
    # Some complicated operation
    ...
except Exception:
    pass
```

- Argh!!!! Boom!

```
try:
    # Some complicated operation
    ...
except Exception:
    # !! TODO
    pass
```

Ariane 5

- Catastrophic failures are often a result of exception handling gone terribly wrong.

Reraising Exceptions

- Log/re-raise

```
try:
    # Some complicated operation
    ...
except Exception as e:
    print("Sorry, it didn't work.")
    print("Reason:", e)
    raise
```

- Useful if you want to do something with the exception, but allow it to propagate

Managing Resources

- Take care to manage system resources correctly

```
def read_csv(filename):
    f = open(filename)
    try:
        ... do whatever ...
    finally:
        f.close()
```

- A more modern version (context manager)

```
def read_csv(filename):
    f = open(filename)
    with f:
        ... do whatever ...
```

- Failure to do this might cause leaky file descriptors, deadlock, or other problems

What Exceptions to Raise?

- Applications should have their own exceptions

```
class ApplicationError(Exception):
    pass

class SomeOtherError(ApplicationError):
    pass
```

- Issue: How do you distinguish between programming mistakes and exceptions that you meant to raise?
- Reserve Python's built-in exceptions for programming mistakes. Catch, don't raise.

Return Codes

- Don't use return codes (usually)

```
def read_csv():
    # Some complicated thing
    ...
    if error:
        return -1   # Oops
    else:
        return result
```

- Return codes are not the "standard" way of signaling errors in Python
- Callers will often forget and program will crash for a different reason later.

Logging

- Use logging for recording diagnostics

```
import logging
log = logging.getLogger(__name__)

def read_csv(filename):
    ...
    try:
        name = row[0]
        shares = int(row[1])
        price = float(row[2])
    except ValueError as e:
        log.warning("Bad row: %s", row)
        log.debug("Reason : %s", e)
```

- Usually a better option than print() functions

Exercise 5.2

Time : 15 Minutes

Object Design

- class statement can define new objects
- There are various OO programming techniques
- Techniques for code reuse (i.e., inheritance)
- But Python is not Java, or C#, or C++, etc..

Don't Use Classes

- Consider this class

```
class TablePrinter(object):
    def __init__(self, formatter):
        self.formatter = formatter
    def output(self, records, fields):
        self.formatter.headings(fields)
        for r in records:
            rowdata = [getattr(r, name) for name in fields ]
            self.formatter.row(rowdata)
```

- Maybe a simple function is good enough

```
def print_table(records, fields, formatter):
    formatter.headings(fields)
    for r in records:
        rowdata = [getattr(r, name) for name in fields ]
        formatter.row(rowdata)
```

- It's okay to make functions. No, really!

Definition Style

- Classes are usually capitalized

```
class Stock(object):
    ...

class TablePrinter(object):
    ...
```

- May include a doc string

```
class Stock(object):
    '''
    Represents a holding of stock
    '''
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

Show the State

- If there's state, make an informative repr()

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

    def __repr__(self):
        return 'Stock(%r, %r, %r)' % (self.name,
                                      self.shares,
                                      self.price)
```

- Simplifies everything (debugging, logging, etc)

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

No Getters/Setters

- Embrace simplicity

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

    def getShares(self):            # NO!
        return self._shares

    def setShares(self, shares):    # NO!
        self._shares = shares
```

- Nobody wants to use extraneous methods
- Use properties/descriptors if you need more

Know the Phrasebook

- There are common ways of using objects

```
for item in obj:        # Iteration
    ...

with obj:               # Context-manager
    ...

value = obj[key]        # Containers/mappings
obj[key] = value
del obj[key]

obj1 + obj2             # Numbers
obj1 - obj2
```

- Your objects should speak the same language
- You'll be happier. So will everyone else.

Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Exercise 5.3

Time : 10 Minutes

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

unittest Module

- Built-in module used for testing
- Used by the standard library
- Commonly used in other applications
- Will briefly illustrate

Example Code

- Suppose you have this function

```
# simple.py
def add(x, y):
    '''
    Adds x and y.
    '''
    return x + y

>>> add(2,2)
4
>>> add('hello','world')
'helloworld'
>>>
```

- Let's test it

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 is near y
self.assertAlmostEqual(x,y,places)

# Assert that an exception is raised
with self.assertRaises(SomeError):
    statement1
    statement2
    ...
```

- 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 % python3 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)

Exercise 5.4

Time : 15 Minutes

Optimization

- Python is interpreted
- Often significantly slower than C
- You may want to optimize your code
- How?

Optimization : Algorithms

- Tip : Understand algorithms
- Don't use an O(N**2) algorithm if there's an O(N) alternative
- A straightforward implementation of a good algorithm is going to beat your tricky optimized version of a bad algorithm

Optimization : Max Speedup

- Tip : Know how your code spends time
- Example: 25% of time is spent in foo()
- You make foo() run ten times as fast
- What is the execution time of the new code?

$$\text{new time} = \underset{\substack{\nearrow \\ \text{unoptimized part}}}{0.75} + \underset{\substack{\nwarrow \\ \text{optimized part}}}{0.25/10} = 0.775 \times \text{old time}$$

- Corollary: Don't optimize unimportant bits

Profiling

- cProfile module
- Collects statistics and prints a report
- Run run it from the command shell

```
bash % python3 -m cProfile someprogram.py
```

Profile Sample Output

```
shell % python3 -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)
...
```

Optimization : Built-in Types

- Tip : Use the built-in datatypes and methods
- Tuples, lists, dicts, sets, etc. are all implemented in C code (and fast)
- Don't implement your own containers (e.g., linked lists, etc.)

Optimization : Layering

- Tip : Avoid Excessive Layering
- Every layer of abstraction has a real cost
- Examples : Descriptors, properties, wrappers, etc.
- Keep it simple

Optimization : Attributes

- Tip : Avoid the (.) attribute lookup operator

```
import math
def foo1(nums):
    for x in nums:
        y = math.sin(x)
```

$\longrightarrow$ 1.00s

```
from math import sin
def foo2(nums):
    for x in nums:
        y = sin(x)
```

$\longrightarrow$ 0.81s

- Every (.) lookup has some cost

Optimization : Locality

- Tip : Frequently accessed items should be made as "local" as possible

```
from math import sin
def foo2(nums):
    for x in nums:
        y = sin(x)
```

$\longrightarrow$ 0.81s

```
def foo3(nums):
    from math import sin
    for x in nums:
        y = sin(x)
```

$\longrightarrow$ 0.72s

- Only difference : local/global variable for sin()

Optimization : Binding

- Tip : Use bound methods

```
def foo1(nums):
    result = []
    for x in nums:                         ⟶ 1.00s
        result.append(sin(x))

def foo2(nums):
    result = []
    result_append = result.append
    for x in nums:                         ⟶ 0.75s
        result_append(sin(x))
```

- You save time on attribute lookup

Optimization : Exceptions

- Tip : Use exceptions for uncommon failures

```
def parse_data(lines):
    for line in lines:
        fields = line.split()
notice this extra check ⟶
        if len(fields) != 3:               ⟶ 1.00s
            continue
        name, shares, price = fields
        ...
```

vs

```
def parse_data(lines):
    for line in lines:
        fields = line.split()              ⟶ 0.88s
        try:
            name, shares, price = fields
catch an exception instead ⟶
        except ValueError:
            continue
```

Optimization : Exceptions

- Followup: How uncommon?

notice this extra check →

```
def parse_data(lines):
    for line in lines:
        fields = line.split()
        if len(fields) != 3:
            continue
        name, shares, price = fields
        ...
```

→ 1.00s

vs

10% of lines with bad data

```
def parse_data(lines):
    for line in lines:
        fields = line.split()
        try:
            name, shares, price = fields
        except ValueError:
            continue
```

→ 1.27s

catch an exception instead →

Optimization : Exceptions

- Tip : Avoid exceptions for common failures

```
try:
    value = items[key]
except KeyError:
    value = None
```

vs

```
if key in items:
    value = items[key]
else:
    value = None
```

or

```
value = items.get(key,None)
```

Exercise 5.5

Time : 10 Minutes

Section 6

Working with Code

Overview

- Advanced function usage/definition
- Introspection
- Code generation (eval, exec)
- Closures
- Callables

What is a function?

- A function is a sequence of statements

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

- Carries out an operation and returns a result

Function Arguments

- Functions operate on passed arguments

```
def func(x,y,z):
    statements
```

arguments

- There are two calling styles

```
a = func(1,2,3)          # Positional arguments
a = func(x=1,y=2,z=3)    # Keyword arguments
```

- You can mix the two styles

```
a = func(1,z=3,y=2)
```

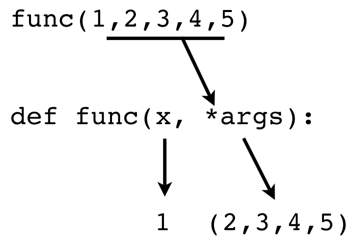
- Positional args always go first. Each argument gets one and only one value

Variable Arguments

- Function that accepts any number of args

```
def func(x, *args):
    ...
```

- Here, the arguments get passed as a tuple



```
func(1,2,3,4,5)

def func(x, *args):

          1   (2,3,4,5)
```

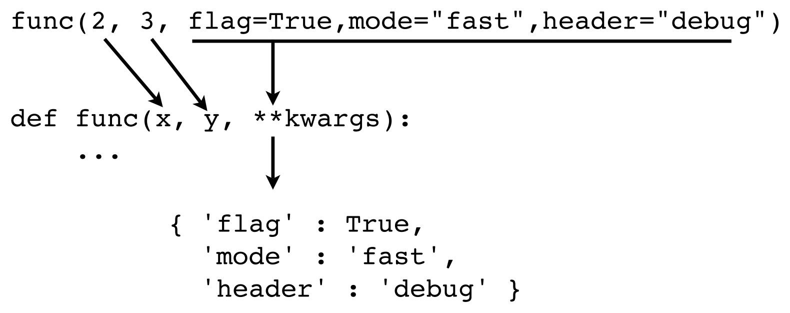
Copyright (C) 2016, http://www.dabeaz.com

Variable Arguments

- Function that accepts any keyword args

```
def func(x, y, **kwargs):
    ...
```

- Extra keywords get passed in a dict



```
func(2, 3, flag=True,mode="fast",header="debug")

def func(x, y, **kwargs):
    ...

          { 'flag' : True,
            'mode' : 'fast',
            'header' : 'debug' }
```

Copyright (C) 2016, http://www.dabeaz.com

Variable Arguments

- A function that takes any arguments

```
def func(*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

Keyword-only Arguments

- Named arguments after (*)

```
def send(msg, *, timeout=None):
    statements

send(msg, 10)           # Error
send(msg, timeout=10)   # OK
```

- Can be mixed with *args, **kwargs

```
def sum(*values, initial=0):
    for val in values:
        initial += val
    return initial

sum(1,2,3, initial=100)      # --> 106
```

- Note: Python 3 only

Passing Tuples and Dicts

- Tuples can expand into function args

```
args = (2,3,4)
func(1, *args)    # Same as func(1,2,3,4)
```

- Dictionaries can expand to keyword args

```
kwargs = {
   'color' : 'red',
   'delimiter' : ',',
   'width' : 400 }

func(data, **kwargs)
# Same as func(data,color='red',delimiter=',',width=400)
```

Exercise 6.1

Time : 20 minutes

Scoping Rules

- Programs assign values to variables

```
x = value           # Global variable

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

- Python stores variables in two scopes
    - Globals (assigned outside functions)
    - Locals (assignments inside functions)
- Scoping rules motivated by C programming

Statement Execution

- All statements execute within two scopes (even statements not part of a function)
- Global scope is always the module in which a function is defined
- Local scope is either private to a function or the same as the global scope (for statements executed at module level)

Local Variables

- All variables assigned in a function 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 read the values of globals

```
x = 42

def func():
    print(x)
```

- However, assignments have no effect

```
def func():
   x = 37

>>> x
42
>>> func()
>>> x
42
>>>
```

Modifying Globals

- If you want to modify a global variable you must declare it as such in the function

```
x = 42

def func():
    global x
    x = 37
```

- global declaration must appear before use
- Only necessary for globals that will be modified (globals are already readable)

globals() and locals()

- globals() - Give you a dictionary representing the contents of global scope
- locals() - Gives you a dictionary representing the contents of local scope
- Can use these to inspect the environment in which a statement will execute

Nested Scopes

- Python allows nested function definitions

```
def countdown(start):
    n = start
    def display():
        print('T-minus %d' % n)
    while n > 0:
        display()
        n -= 1
```

- In such definitions, inner functions can freely read the value of local variables defined in outer functions
- An exception to the two-scope rule

Nested Scopes

- Inner functions can modify outer variables

```
def countdown(start):
    n = start
    def display():
        print('T-minus %d' % n)
    def decrement():
        nonlocal n
        n -= 1
    while n > 0:
        display()
        decrement()
```

- Must be declared as "nonlocal"
- Only in Python 3. Usage is rare.

Exercise 6.2

Time : 15 minutes

Function Objects

- When you define a function, the function itself becomes a kind of object that you can manipulate
- Can assign to variables, place in containers, pass around as data, etc.
- Functions can also be inspected

Documentation Strings

- First line of function may be string

```
def func(a, b):
    'This function does something.'
    ...
```

- The doc string is stored in __doc__

```
>>> func.__doc__
'This function does something.'
>>>
```

- Help tools look at __doc__, but other programs might examine it for various purposes (testing, etc.)

Annotations

- Arguments and return might be annotated

```
def func(a:int, b:int) -> int:
    ...
```

- Stored in __annotations__

```
>>> func.__annotations__
{'a': <class 'int'>, 'b': <class 'int'>,
 'return': <class 'int'>}
>>>
```

- Annotations do nothing--purely informational for other code that might want to look at them.

Function Attributes

- Little known fact : You can attach arbitrary attributes to a function

```
def func(a, b):
    'This function does something.'
    ...

func.threadsafe = False
func.blah = 42
```

- Under the covers, each function has a dictionary (__dict__) that holds these values
- Useful for code that manipulates functions

Function Inspection

- Almost every aspect of a function can be inspected if you know where to look

```
def func(a, b, c=42):
    'This function does something.'
    ...
>>> func.__name__
'func'
>>> func.__defaults__
(42,)
>>> func.__code__
<code object func at 0x325f50, file "<stdin>", line 1>
>>> func.__code__.co_argcount
3
>>> func.__code__.co_varnames
('a', 'b', 'c')
```

- Use the dir() function to see more

inspect Module

- Use the inspect module to get details about functions in a more useful form

```
def func(a, b, c=42):
    ...

>>> import inspect
>>> sig = inspect.signature(func)
>>> print(sig)
(a, b, c=42)
>>> list(sig.parameters)
['a', 'b', 'c']
>>> sig.parameters['c'].default
42
>>>
```

- Many more module features (not shown)

Exercise 6.3

Time : 15 minutes

eval() and exec()

- eval(code) - Evaluates an expression

```
>>> x = 10
>>> eval('3*x - 2')
28
>>>
```

- exec(code) - Executes arbitrary statements

```
>>> exec('for i in range(5): print(i)')
0
1
2
3
4
>>>
```

- Code executes in current globals()/locals()

eval() and exec()

- Caution: Modifications to local scope are lost

```
def func():
    x = 10
    exec('x = 15; print(x)')   # ---> 15
    print(x)                   # ---> 10    ?????
```

- eval(*expr* [, *globals* [, *locals*]])
- exec(*code* [, *globals* [, *locals*]])

```
def func():
    x = 10
    loc = locals()
    exec('x = 15; print(x)', globals(), loc)  # ---> 15
    x = loc['x']
    print(x)                                  # ---> 15
```

eval/exec Caution

- Use these features with extreme care
- Overuse can lead to various problems
- Code doesn't run as fast as normal functions
- Bizarre interaction with scoping/variables

Exercise 6.4

Time : 15 minutes

Functional Programming

- A programming style relying primarily on functions and evaluation of expressions
- A few general properties
  - Function evaluation only
  - No side effects, no mutable state
  - Higher order functions
- Prior example: List comprehensions

Higher Order Functions

- Essential features...
  - Functions can accept functions as input
  - Functions can return functions as results
- Python supports both

Functions as Input

- Functions can be passed as inputs
- Example:

```
def map(func, items):
    return [func(x) for x in items]

def square(x):
    return x * x

nums = [1, 2, 3, 4]
result = map(square, nums)  # [1, 4, 9, 16]
```

- A function is just like any other bit of data

Returning Functions

- Consider the following function

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

- A function that returns another function?

```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
7
>>>
```

- Notice that it works, but ponder it...

Nested Scopes (Reprise)

- Observe how the inner function refers to variables defined by the outer function

```
def add(x,y):
    def do_add():
        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()
7
```

Where are the x,y 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():
        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
- Common applications:
    - Delayed/Deferred evaluation
    - Partial function application
    - Code creation ("macros")

Delayed Evaluation

- Go back to our original example

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

- This is an example of "delayed evaluation"
- add() doesn't do anything, it returns a function that carries out work later

```
>>> a = add(3,4)
>>> a
<function do_add at 0x6a670>
>>> a()
7
```

Delayed Evaluation

- Delayed evaluation defers calculations until a later point in program execution
- Perhaps the calculation is meant to execute in response to a future event
- Perhaps the calculation can't be performed right this moment because all of the needed inputs aren't available yet

Exercise 6.5

Time : 25 Minutes

Callable Objects

- You can define your own objects that emulate Python functions (e.g., "callables")

```
class Callable(object):
    def __call__(self,*args,**kwargs):
        print('Calling', args, kwargs)
```

- Must implement __call__ special method

```
>>> c = Callable()
>>> c(2,3,color='red')
Calling (2, 3) {'color': 'red'}
>>>
```

- Free to do anything you want in __call__

Defining Callables

- Callable objects sometimes defined when you need to have more than just a function (e.g., storing extra data, caching, etc.)

```
class Memoize(object):
    def __init__(self,func):
        self._cache = {}
        self._func = func
    def __call__(self,*args):
        if args in self._cache:
           return self._cache[args]
        r = self._func(*args)
        self._cache[args] = r
        return r
    def clear(self):
        self._cache.clear()
```

Exercise 6.6

Time : 10 Minutes

Section 7

Metaprogramming

Introduction

- Writing programs where there is a lot of code replication is usually problematic
- Tedious to write
- Hard to maintain
- Painful if you decide that you need to make a change (or fix a bug) in all of that extremely repetitive code

Metaprogramming

- Metaprogramming pertains to the problem of writing code that manipulates other code
- Common examples:
  - Macros
  - Wrappers
  - Aspects
- Essentially, it's doing things to code

Python Metaprogramming

- Major features
  - Decorators
  - Class decorators
  - Metaclasses
- We're going to talk about all of them
- They are not as difficult to grasp as you think

Decorators

- A decorator is a function that creates a wrapper around another function
- The wrapper is a new function that works exactly like the original function (same arguments, same return value) except that some kind of extra processing is carried out
- Let's see a simple example first

Wrapper Functions

- Here is a simple function:

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

- Here is an example of a wrapper function

```
def logged_add(x, y):
    print('Calling add')
    return add(x, y)
```

- Example use:

```
>>> add(3, 4)
7
>>> logged_add(3, 4)
Calling add
7
>>>
```

This extra output is created by the wrapper, but the original function is still called to get the result

Creating Wrappers

- Insight : You can write a function that makes a wrapper around any function

```
def logged(func):
    # Define a wrapper function around func
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

- Usage:

```
>>> logged_add = logged(add)
>>> logged_add
<function wrapper at 0x378670>
>>> logged_add(3, 4)
Calling add
7
>>>
```

Wrappers as Replacements

- When you create a wrapper, you often want to replace the original function with it

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

# Replace add with a wrapped version
add = logged(add)
```

- Other code continues to use the original function name, but is unaware that a wrapper has been injected (that's the whole point)

```
>>> add(3, 4)
Calling add
7
>>>
```

Decorator Concept

- When you replace a function with a wrapper, you are usually giving the function extra functionality
- This process is known as "decoration"
- You are "decorating" a function with some extra features

Decorator Syntax

- The definition of a function and wrapping almost always occur together

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

- However, it looks weird and is error prone
- The @decorator syntax simplifies it

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

Decorator Syntax

- Whenever you see a decorator, a function is getting wrapped. That's it
- Example : In classes

```
class Foo(object):
    @staticmethod
    def bar():
        ...
    @classmethod
    def spam(cls):
        ...
    @property
    def name(self):
        ...
```

$\longrightarrow$

```
class Foo(object):
    def bar():
        ...
    bar = staticmethod(bar)
    def spam(cls):
        ...
    spam = classmethod(spam)
    def name(self):
        ...
    name = property(name)
```

Using Decorators

- Use a decorator anytime you want to define a kind of "macro" involving function definitions
- There are many possible applications
  - Debugging and diagnostics
  - Avoiding code replication
  - Enabling/disabling optional features

Timing Measurements

- A decorator that reports execution time

```
import time
def timethis(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end - start)
        return r
    return wrapper
```

- Usage:

```
@timethis
def bigcalculation():
    statements
    statements
```

Exercise 7.1

Time : 15 Minutes

Advanced Decorators

- There are a few tricky additional details
  - Multiple decorators
  - Decorators and metadata
  - Decorators with arguments

Multiple Decorators

- You can apply as many decorators as you want

```
@foo
@bar
@spam
def add(x, y):
    return x + y
```

- This is the same as this:

```
add = foo(bar(spam(add)))
```

- To keep your sanity, it's probably not a good idea to go overboard with it

Function Metadata

- When you define a function, there is some extra information stored (name, doc strings, etc.)

```
def add(x, y):
    'Adds x and y'
    return x + y

>>> add.__name__
'add'
>>> add.__doc__
'Adds x and y'
>>> help(add)
Help on function add in module __main__:

add(x, y)
    Adds x and y
>>>
```

The Metadata Problem

- Decorators don't preserve metadata

```
@logged
def add(x, y):
    'Adds x and y'    return x + y

>>> add.__name__
'wrapper'
>>> add.__doc__
>>> help(add)
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
>>>
```

- This is a problem

Copying Metadata

- Decorators should copy metadata

manual copying of metadata →

```
def logged(func):
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    wrapper.__name__ = func.__name__
    wrapper.__doc__ = func.__doc__
    return wrapper
```

- A better solution : use @wraps

Copies metadata from func to the wrapper →

```
from functools import wraps

def logged(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

Decorators with Args

- Decorators can accept arguments

```
@decorator(x, y, z)
def func():
    ...
```

- It's mind boggling, but here's what happens

```
def func():
    ...

func = decorator(x, y, z)(func)
```

- The decorator function must return a <u>function</u> which is called to make a wrapper

Decorators with Args

- Example: Logging with a custom message

```
def logmsg(message):
    def logged(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(message.format(name=func.__name__))
            return func(*args, **kwargs)
        return wrapper
    return logged
```

- Example use:

```
@logmsg('You called {name}')
def add(x, y):
    return x + y
```

Decorators with Args

- Example: Logging with a custom message

```
def logmsg(message):
    def logged(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(message.format(name=func.__name__))
            return func(*args, **kwargs)
        return wrapper
    return logged
```

Outer function takes the arguments

Arguments can be used by the code inside

- The outer function is like an enclosing environment that gets added to the decorator to accept the extra arguments

Decorators with Args

- Example: Logging with a custom message

The same decorator code as before

```
def logmsg(message):
    def logged(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(message.format(name=func.__name__))
            return func(*args, **kwargs)
        return wrapper
    return logged
```

- Inner functions are standard decorator code

Decorators with Args

- Decorators with args are more general
- You can specialize to a no-argument case

```
logged = logmsg('Calling {name}')

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

- This is subtle, but useful for simplifying code

Exercise 7.2

Time : 15 Minutes

Class Decorators

- Decorators can be applied to class definitions

```
@decorator
class Foo(object):
    def bar(self):
        ...
    def spam(self):
        ...
```

- It's exactly the same as doing this:

```
class Foo(object):
    def bar(self):
        ...
    def spam(self):
        ...
Foo = decorator(Foo)
```

- Manipulates or wraps a class

Class Decorators

- Most class decorators inspect or do something special with the class definition
- Typical prototype

```
def decorator(cls):
    # Do something with cls
    ...
    # Return the original class back
    return cls
```

- Observe: The original class is not replaced

Example

- Recording all attribute lookups

```
def logged_getattr(cls):
    # Get the original implementation
    orig_getattribute = cls.__getattribute__

    # Replacement method
    def __getattribute__(self, name):
        print('Getting:', name)
        return orig_getattribute(self, name)

    # Attach to the class
    cls.__getattribute__ = __getattribute__
    return cls
```

- This is replacing a method of the class

Example

```
@logged_getattr
class Spam(object):
    def foo(self):
        pass
    def bar(self):
        pass

>>> s = Spam()
>>> s.x = 23
>>> s.x
Getting: x
>>> s.foo()
Getting: foo
>>> s.bar()
Getting: bar
>>>
```

Notice how all lookups now have logging

Exercise 7.3

Time : 15 Minutes

Disclaimer

- What follows is Python's most advanced bit
- Took years for programmers to even understand what anyone was talking about
- Even now, the details are somewhat hairy
- Also known as the "killer joke"

Types

- As you hopefully know, all values in Python have an associated type
- Example:

```
>>> x = 42
>>> type(x)
<class 'int'>
>>> s = 'Hello'
>>> type(s)
<class 'str'>
>>> items = [1,2,3]
>>> type(items)
<class 'list'>
>>>
```

Type Constructor

- The "type" is usually a callable for creating objects of that type
- Example:

```
>>> items = [1,2,3]
>>> type(item)
<class 'list'>
>>> a = list()        # Create a new list object
>>> a
[]
>>> b = tuple(items)  # Convert to a tuple
>>>
```

- Type conversions like this are common

Types and Classes

- Classes also define new types

```
class Foo(object):
    pass

>>> f = Foo()
>>> type(f)
<class '__main__.Foo'>
>>>
```

- It is exactly the same as with built-ins
- The class is the type of instances created
- The class is a callable that creates instances

Types of Classes

- Classes are instances of types
- Observe by getting the type of a class itself

```
>>> class Foo(object):
...     pass
...
>>> type(Foo)
<class 'type'>
>>> isinstance(Foo,type)
True
>>>
```

Recall: type() tells you the type of an object. Here we're using it on a class itself.

- This requires some thought, but it should make some sense (a class is simply a type)

Creating Types

- Head explosion: types are represented by their own class (type)

```
class type(object):
    ...

>>> type
<class 'type'>
>>>
```

- This class creates new "type" objects
- In fact, this is the class that processes class definitions when you define your own classes

Classes Deconstructed

- Consider a class:

```
class Foo(object):
    def __init__(self,name):
        self.name = name
    def bar(self):
        print("I'm Foo.bar")
```

- What are its components?
  - Name ("Foo")
  - Base classes (object)
  - Functions (__init__,bar)

Creating a Class

- You can create a class manually from pieces

```
# Define some method functions
def __init__(self,name):
    self.name = name
def bar(self):
    print("I'm Foo.bar")

# Make a method table
methods = {'__init__': __init__,
           'bar': bar }

# Make a new type (Foo)
Foo = type('Foo', (object,), methods)
```

- These steps mimic the class statement

Class Definition Process

- What happens during class definition?

```
class Foo(object):
    def __init__(self,name):
        self.name = name
    def bar(self):
        print("I'm Foo.bar")
```

- Step1: Body of class is captured

```
body = '''
   def __init__(self,name):
        self.name = name
   def bar(self):
        print("I'm Foo.bar")
'''
```

Class Definition Process

- Step 2: A dictionary is created

```
__dict__ = type.__prepare__('Foo', (object,))
```

- Normally, you get a plain Python dictionary

```
>>> type.__prepare__('Foo', (object,))
{}
>>>
```

- Some extra metadata is inserted

```
__dict__['__qualname__'] = 'Foo'
__dict__['__module__'] = 'modulename'
```

Class Definition Process

- Step 3: Class body is executed in the dict

```
exec(body, globals(), __dict__)
```

- The statements in the body run like a script
- Afterwards, __dict__ is populated

```
>>> __dict__
{
  '__init__': <function __init__ at 0x4da10>,
  'bar': <function bar at 0x4dd70>},
  '__qualname__': 'Foo',
  '__module__': 'modulename'
}
>>>
```

Class Definition Process

- Step 4: Class is constructed from its name, base classes, and the dictionary

```
>>> Foo = type('Foo', (object,), __dict__)
>>> Foo
<class '__main__.Foo'>
>>> f = Foo('Guido')
>>> f.bar()
I'm Foo.bar
>>>
```

- type(name, bases, dict) constructs a class object

Exercise 7.4

Time : 15 Minutes

Metaclasses Defined

- A class that creates classes is called a metaclass
- type is an example of a metaclass

The Metaclass Hook

- Python provides a hook that allows you to override the class creation steps
- You can use a different metaclass than "type"
- Using this, you can completely customize what happens when a class is created.

Metaclass Selection

- metaclass keyword argument
- Sets the class used to create the class object

```
class Foo(metaclass=type):
   def __init__(self, name):
       self.name = name
   def bar(self):
       print("I'm Foo.bar")
```

- By default, it's set to 'type', but you change it to something else (more shortly)

Metaclass Selection

- Compatibility note: Python 2 is different
- Use the __metaclass__ attribute instead

```
class Foo:
   __metaclass__ = type         # Python 2 only
   def __init__(self, name):
       self.name = name
   def bar(self):
       print("I'm Foo.bar")
```

- Comment: Very difficult to provide Py2/3 compatibility due to syntax difference

Metaclass Inheritance

- If no metaclass is set, Python uses the same type as the base class

```
class Foo(object):
   def __init__(self, name):
       self.name = name
   def bar(self):
       print("I'm Foo.bar")

>>> object.__class__
<class 'type'>
>>> type(Foo)
<class 'type'>
```

- Note: this is why you rarely see it

Creating a New Metaclass

- You inherit from type and redefine methods such as __new__, __prepare__, etc.

```
class mytype(type):
    @staticmethod
    def __new__(meta, name, bases, methods):
        print('Creating class : ', name)
        print('Base classes   : ', bases)
        print('Attributes     : ', list(methods))
        return super().__new__(meta, name, bases, methods)
```

- Then you define a new root-object

```
class myobject(metaclass=mytype):
    pass
```

Using a Metaclass

- To use the new metaclass, define classes so that they inherit from your root object

```
class Foo(myobject):
    def __init__(self, name):
        self.name = name
    def bar(self):
        print("I'm Foo.bar")
```

- You should see your metaclass at work

```
Creating class :  Foo
Base classes   :  (<class '__main__.myobject'>,)
Attributes     :  ['bar', '__module__', '__init__']
```

Exercise 7.5

Time : 10 Minutes

Typical Applications

- Enforcing coding conventions
- Debugging and diagnostics
- Automatic wrapping of methods (e.g., applying a decorator to all methods)
- Filling in missing details

Using a Metaclass

- Metaclasses allow class definitions to be monitored and manipulated
- There are 4 main interception points

```
type.__prepare__(name, bases)
                 ↓
type.__new__(type, name, bases, dict)
                 ↓
type.__init__(cls, name, bases, dict)
```

class definition

```
type.__call__(cls, *args, **kwargs)
```

instance creation

Example: Duplicate Check

```
class dupedict(dict):
    def __setitem__(self, key, value):
        assert key not in self, '%s duplicated' % key
        super().__setitem__(key, value)

class dupemeta(type):
    @classmethod
    def __prepare__(cls, name, bases):
        return dupedict()
```

- Example:

```
class A(metaclass=dupemeta):
    def bar(self):
        pass
    def bar(self):      ⟶ Fails! Duplicate
        pass
```

Example: Decoration

```
def decorator(func):
    ...
    # Decorator
    ...

class meta(type):
    @staticmethod
    def __new__(meta, clsname, bases, dict):
        for key, val in dict.items():
            if callable(val):
                dict[key] = decorator(val)
        return super().__new__(meta, clsname, bases, dict)
```

- This class wraps all methods with a decorator

Example: Class Registration

```
class meta(type):
    _registered = { }
    def __init__(cls, clsname, bases, dict):
        super().__init__(clsname, bases, dict)
        meta._registered[clsname] = cls
```

- This tracks all subclasses

```
class A(metaclass=meta):
    pass

class B(A):
    pass

class C(B):
    pass
```

```
>>> meta._registered
{'A': <class '__main__.A'>,
'B': <class '__main__.B'>,
'C': <class '__main__.C'>}
>>>
```

Example: Instance Creation

```
class meta(type):
    def __call__(cls, *args, **kwargs):
        print('Creating instance of', cls)
        return super().__call__(*args, **kwargs)
```

- Example:

```
>>> class A(metaclass=meta):
        pass

>>> a = A()
Creating instance of <class '__main__.A'>
>>>
```

- Potentially useful for special cases (singletons, caching, etc.)

Commentary

- Metaclasses are not something you should be defining without really good reasons
- Target audience:
    - Framework builders
    - Library developers
- End users should not be messing around with metaclasses in their own code

Exercise 7.6

Time : 20 Minutes

Section 8

Iterators, Generators, and Coroutines

Iteration

- Iteration defined: 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: Protocol

- Iteration

```
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
>>>
```

Generators

- Generators simplify customized iteration

```
def countdown(n):
    print('Counting down from', n)
    while n > 0:
        yield n
        n -= 1

>>> for i in countdown(5):
...     print('T-minus', i)
...
Counting down from 5
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
>>>
```

Generator Functions

- Behavior is 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>
>>> next(x)    # invokes 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()

```
>>> next(x)
9
>>> next(x)
8
>>>
```

Generator Functions

- When the generator returns, iteration stops

```
>>> next(x)
1
>>> next(x)
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.

Iterable Objects

- Objects that implement iteration should almost always use generators

```
class Countdown(object):
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        n = self.n
        while n > 0:
            yield n
            n -= 1

    def __reversed__(self):
        n = 1
        while n <= self.n:
            yield n
            n += 1
```

generator

Iterable Objects

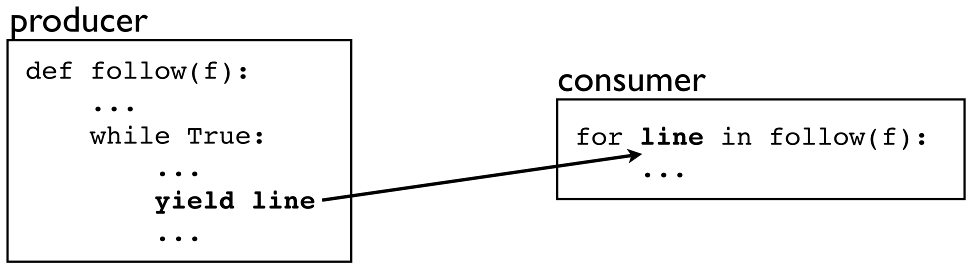
```
>>> c = Countdown(5)
>>> for n in c:
...     print(n)
...
5
4
3
2
1
>>> for n in reversed(c):
...     print(n)
...
1
2
3
4
5
>>>
```

Exercise 8.1

Time : 20 Minutes

Producers & Consumers

- Generators are closely related to various forms of "producer-consumer" programming



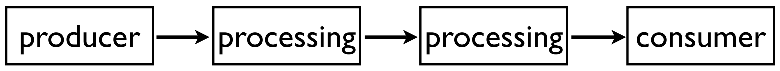
```
def follow(f):
    ...
    while True:
        ...
        yield line
        ...
```

```
for line in follow(f):
    ...
```

- yield produces values
- for consume values

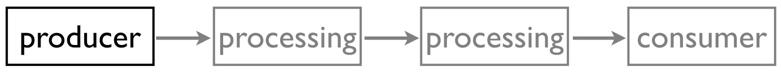
Generator Pipelines

- 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

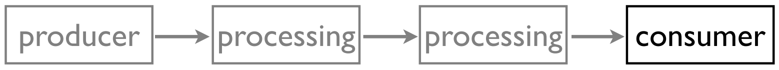
Generator Pipelines



```
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

Generator Pipelines

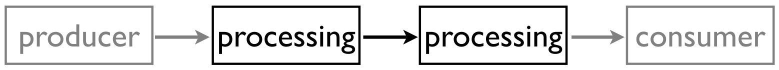


```
def producer():
    ...
    yield item
    ...
```

```
def consumer(s):
    for item in s:
        ...
```

- Consumer is just a simple for-loop
- It gets items and does something with them

Generator Pipelines



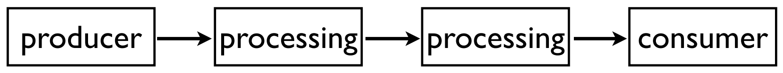
```
def producer():
    ...
    yield item
    ...
```

```
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)

Generator Pipelines



```
def producer():          def processing(s):          def consumer(s):
    ...                      for item in s:              for item in s:
    yield item                   ...                         ...
    ...                          yield newitem
                                 ...
```

- Pipeline setup (in your program)

```
a = producer()

b = processing(a)

c = consumer(b)
```

- You will notice that data incrementally flows through the different functions

Exercise 8.2

Time : 15 minutes

Yield as an Expression

- In generators, yield can be used as an *expression*
- For example, on the right side of an assignment

```
def match(pattern):
    print('Looking for %s' % pattern)
    while True:
        line = yield
        if pattern in line:
            print(line)
```

- Question : What is its value?

Coroutines

- If you use yield like this, you get a "coroutine"
- It defines a function to which you send values

```
>>> g = match('python')
>>> next(g)              # Prime it (explained shortly)
Looking for python
>>> g.send('Yeah, but no, but yeah, but no')
>>> g.send('A series of tubes')
>>> g.send('python generators rock!')
python generators rock!
>>>
```

- Sent values are returned by (yield)

Coroutine Execution

- Execution is the same as for a generator
- When you call a coroutine, nothing happens
- They only run in response to next() and send() methods

```
>>> g = match('python')
>>> next(g)
Looking for python
>>>
```

Notice that no output was produced

On first operation, coroutine starts running

Coroutine Priming

- All coroutines must be "primed" by first calling next() (or send(None))
- This advances execution to the location of the first yield expression.

```
def match(pattern):
    print('Looking for %s' % pattern)
    while True:
        line = yield
        if pattern in line:
            print(line)
```

next() advances the coroutine to the first yield expression

- At this point, it's ready to receive a value

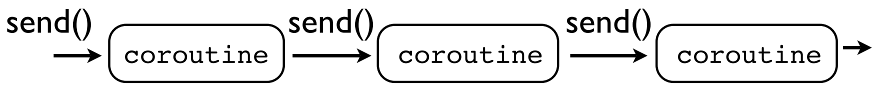
Using a Decorator

- Remembering to call next() is easy to forget
- Solved by wrapping coroutines with a decorator

```
def consumer(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

@consumer
def match(pattern):
    ...
```

Processing Pipelines

- Coroutines can also be used to set up pipes



- You just chain coroutines together and push data through the pipe with send() operations

An Example

- A source that mimics Unix 'tail -f'

```
import time
def follow(filename, target):
    f = open(filename)
    f.seek(0,2)      # Go to the end of the file
    while True:
         line = f.readline()
         if line != '':
             target.send(line)
         else:
             time.sleep(0.1)    # Sleep briefly=
```

- A consumer that just prints the lines

```
@consumer
def printer():
    while True:
         line = yield
         print(line, end=' ')
```

An Example

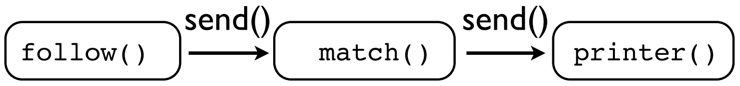
- A filter coroutine

```
@consumer
def match(pattern, target):
    while True:
        line = yield           # Receive a line
        if pattern in line:
            target.send(line)    # Send to next stage
```

- Hooking it up

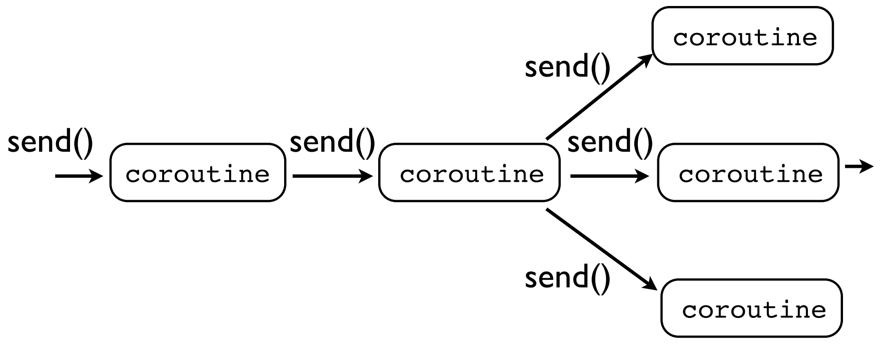
```
follow('access-log',
       match('python',
       printer()))
```

- A picture



Dataflow

- With coroutines, you can "fan out"



- More possibilities than a simple pipeline

Exercise 8.3

Time : 15 Minutes

Generator Control Flow

- Generators have support for forced termination and exception handling
  - .close() method - terminates
  - .throw() method - raise an exception
- Examples follow

Closing a Generator

- Use .close() method to shutdown

```
g = genfunc()     # A generator
...
g.close()
```

- This raises GeneratorExit at yield

```
def genfunc():
    ...
    try:
        yield item
    except GeneratorExit:
        # .close() was invoked
        # perform cleanup (if any)
        ...
        return
```

Raising Exceptions

- Use .throw(type [,val [, tb]]) for exceptions

```
g = genfunc()     # A generator
...
g.throw(RuntimeError, "You're dead")
```

- This raises an exception at the yield

```
def genfunc():
    ...
    try:
        yield item
    except RuntimeError as e:
        # Handle the exception
        ...
```

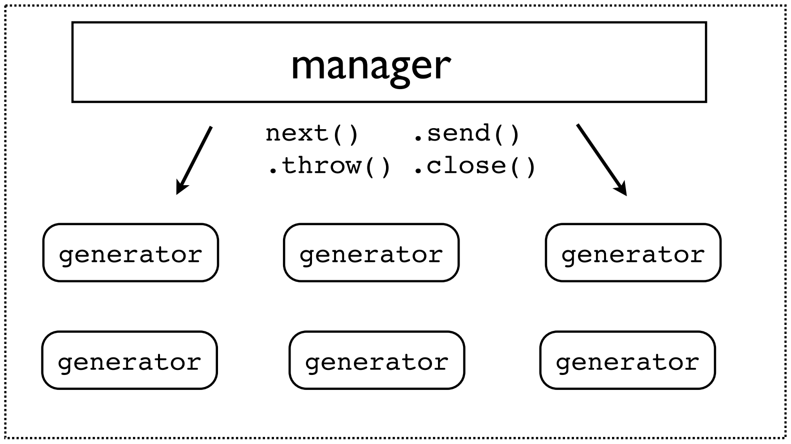
Exercise 8.4

Time : 10 Minutes

Managed Generators

- Observation: A generator function can not execute solely by itself. It must be driven by something else (e.g., for-loop, send(), etc)
- Observation: The yield statement represents a point of preemption. Generators suspend at the yield and don't resume until instructed.

Managed Generators



- Idea: A manager will coordinate the execution of a collection of executing generators

Managed Generators

- Typical applications
  - Concurrency (tasklets, greenlets, etc.)
  - Actors
  - Event simulation
- This is a big topic
- Will give a simple example

Example : Concurrency

- Define some "task" functions

```
def countdown(n):
    while n > 0:
        print('T-minus', n)
        yield
        n -= 1

def countup(n):
    x = 0
    while x < n:
        print('Up we go', x)
        yield
        x += 1
```

- Carefully observe: just a bare "yield"

Example : Concurrency

- Instantiate some tasks in a queue

```
tasks = deque([
    countdown(10),
    countdown(5),
    countup(20)
])
```

- Run a little scheduler (the manager)

```
def run():
    while tasks:
        t = tasks.popleft()      # Get a task
        try:
            next(t)              # Run to yield
            tasks.append(t)      # Reschedule
        except StopIteration:
            pass
```

Example : Concurrency

- Output

```
T-minus 10
T-minus 5
Up we go 0
T-minus 9
T-minus 4
Up we go 1
T-minus 8
T-minus 4
Up we go 2
...
```

- We see tasks cycling, but there are no threads

Exercise 8.5

Time : 20 Minutes

Delegating Generation

- Problem: Library functions involving generators
- Example: Generator Chaining

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

g1 = countdown(5)       # 5 4 3 2 1
g2 = countdown(10)      # 10 9 8 7 6 5 4 3 2 1

g = chain(g1, g2)       # 5 4 3 2 1 10 9 8 7 6 5 4 3 2 1
```

- Problem: How do you write the chain() func?

Delegating Generation

- Option 1: Drive the generator yourself

```
def chain(g1, g2):
    for x in g1:
        yield x
    for x in g2:
        yield x
```

- You need to manually control each generator with for-loops, send(), throw(), etc.
- Can get quite complicated for coroutines

Delegating Generation

- Option 2: Let Python drive it (yield from)

```
def chain(g1, g2):
    yield from g1
    yield from g2
```

- Whatever code normally drives the generators will run it for you
- Greatly simplifies managed generators
- Note: Requires Python 3.3 or newer.

More Information

- "Generator Tricks for Systems Programmers" tutorial from PyCon'08

  http://www.dabeaz.com/generators

- "A Curious Course on Coroutines and Concurrency" tutorial from PyCon'09

  http://www.dabeaz.com/coroutines

- "Generators: The Final Frontier" tutorial from PyCon'14

  http://www.dabeaz.com/finalgenerator

Exercise 8.6

Time : 15 Minutes

Section 9

Modules and Packages

Introduction

- You've written some code
- Now you need to organize it
- Possibly give it to others
- How do you do it?

Modules Revisited

- As you know, every 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')
...
```

Module Objects

- Modules are objects

```
>>> import foo
>>> foo
<module 'foo' from 'foo.py'>
>>>
```

- A "namespace" for definitions inside

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

- Actually a layer on top of a dictionary (globals)

```
>>> foo.__dict__['grok']
<function grok at 0x1006b6c80>
>>>
```

Special Variables

- A few special variables defined in a module

```
__file__      # Name of the source file
__name__      # Name of the module
__doc__       # Module documentation string
```

- Example: "main" check

```
if __name__ == '__main__':
   print('Running as the main program')
else:
   print('Imported as a module using import')
```

Import Implementation

- Import in a nutshell (pseudocode)

```
import types

def import_module(name):
    # locate the module and get source code
    filename = find_module(name)
    code = open(filename).read()

    # Create the enclosing module object
    mod = types.ModuleType(name)

    # Run it
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

- Source is exec'd in module dictionary
- Contents are whatever is left over

Module Cache

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

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

Import Caching

- Import (pseudocode)

```
import types
import sys

def import_module(name):
    # Check for cached module
    if name in sys.modules:
        return sys.modules[name]

    filename = find_module(name)
    code = open(filename).read()
    mod = types.ModuleType(name)
    sys.modules[name] = mod

    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

- There is more, but this is basically it

from module import

- Selected symbols can be imported locally

```
# bar.py
from foo import grok

grok(2)
```

- Useful for frequently used names
- Confusion: This does not change how import works. The entire module executes and is cached. This merely copies a name.

```
grok = sys.modules['foo'].grok
```

from module import *

- Takes all symbols from a module and places them into the caller's namespace

```
# bar.py
from foo import *

grok(2)
spam('Hello')
...
```

- However, it only applies to names that don't start with an underscore (_)
- _*name* often used when defining non-imported values in a module.

Module Reloading

- Modules can sometimes be reloaded

```
>>> import foo
...
>>> import importlib
>>> importlib.reload(foo)
<module 'foo' from 'foo.py'>
>>>
```

- It re-executes the module source on top of the already defined module dictionary

```
# pseudocode
def reload(mod):
    code = open(mod.__file__, 'r')
    exec(code, mod.__dict__, mod.__dict__)
    return mod
```

Module Reloading Danger

- Module reloading is not advised
- Problem: Existing instances of classes will continue to use old code after reload
- Problem: Doesn't update definitions loaded with 'from module import name'
- Problem: Likely breaks code that performs typechecks or uses super()

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/python35.zip',
 '/usr/local/lib/python3.5',
 '/usr/local/lib/python3.5/plat-darwin',
 '/usr/local/lib/python3.5/lib-dynload',
 '/usr/local/lib/python3.5/site-packages']
```

Module Search Path

- sys.path contains search path
- Can manually adjust if you need to

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

- Paths also added via environment variables

```
% env PYTHONPATH=/project/foo/pyfiles python3
Python 3.5.0 (default, Oct 27 2015, 13:20:23)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)]
>>> import sys
>>> sys.path
['', '/project/foo/pyfiles',
 '/usr/local/lib/python35.zip', ... ]
```

Exercise 9.1

10 minutes

Organizing Libraries

- It is standard practice for Python libraries to be organized as a hierarchical set of modules that sit under a top-level package name

```
packagename
packagename.foo
packagename.bar
packagename.utils
packagename.utils.spam
packagename.utils.grok
packagename.parsers
packagename.parsers.xml
packagename.parsers.json
...
```

- Other programming languages have a similar convention (e.g., Java)

Naming Conventions

- It is standard practice for package and module names to be concise and lowercase

```
foo.py        not        MyFooModule.py
```

- Use a leading underscore for modules that are meant to be private or internal

```
_foo.py
```

- Don't use names that match common standard library modules (confusing)

```
packagename/
        math.py
```

Flat vs. Deep

- As a general rule, Python programmers tend to prefer flat namespaces

```
import packagename.foo
```

- As opposed to deep hierarchies

```
import dabeaz.projects.packagename.libraries.util.foo
```

- It's not always possible in practice, but use common sense (don't over-engineer it and try to keep it as simple as it needs to be)

Creating a Package

- To create the module library hierarchy, organize files on the filesystem in a directory with the desired structure

```
packagename/
            foo.py
            bar.py
            utils/
                  spam.py
                  grok.py
            parsers/
                  xml.py
                  json.py
...
```

Creating a Package

- Add __init__.py files to each directory

```
packagename/
            __init__.py
            foo.py
            bar.py
            utils/
                  __init__.py
                  spam.py
                  grok.py
            parsers/
                  __init__.py
                  xml.py
                  json.py
...
```

- These can be empty, but they should exist

Using a Package

- Once you have the __init__.py files, the import statement should just "work"

```
import packagename.foo
import packagename.parsers.xml

from packagename.parsers import xml
```

- *Almost* everything should work the same way that it did before except that import statements now have multiple levels

Fixing Relative Imports

- Relative imports of submodules don't work

```
spam/
     __init__.py
     foo.py
     bar.py
```

```
# bar.py
import foo   # Fails (not found)
```

- The issue: Resolving name clashes between top-level packages and submodules

```
spam/
     __init__.py
     os.py
     bar.py
```

```
# bar.py
import os   # ??? (uses stdlib)
```

- imports are always "absolute" (from top level)

Package Relative Imports

- Consider a package

```
spam/
     __init__.py
     foo.py
     bar.py
     grok/
         __init__.py
         blah.py
```

- Package relative imports

```
# bar.py

from . import foo         # Imports ./foo.py
from .foo import name     # Load a specific name

from .grok import blah    # Imports ./grok/blah.py
```

Package Environment

- Packages define a few useful variables

```
__package__         # Name of the enclosing package
__path__            # Search path for subcomponents
```

- Example:

```
>>> import xml
>>> xml.__package__
'xml'
>>> xml.__path__
['/usr/local/lib/python3.5/xml']
>>>
```

- Useful if code needs to obtain information about its enclosing environment

Exercise 9.2

10 minutes

__init__.py Usage

- What are you supposed to do in those files?
- Main use: stitching together multiple source files into a "unified" top-level import

Module Assembly

- Consider two submodules in a package

```
spam/
     foo.py
```

```
# foo.py

class Foo(object):
    ...
    ...
```

```
     bar.py
```

```
# bar.py

class Bar(object):
    ...
    ...
```

- Suppose you wanted to combine them

Module Assembly

- Combine in __init__.py

```
spam/
     foo.py
```

```
# foo.py

class Foo(object):
    ...
    ...
```

```
     bar.py
```

```
# bar.py

class Bar(object):
    ...
    ...
```

```
__init__.py
```

```
# __init__.py

from .foo import Foo
from .bar import Bar
```

Module Assembly

- Users see a single unified top-level package

```
import spam

f = spam.Foo()
b = spam.Bar()
...
```

- Split across submodules is hidden

Case Study

- The collections "module"
- It's actually a package with a few components

_collections.so

```
deque
defaultdict
```

_collections_abc.py

```
Container
Hashable
Mapping
...
```

collections/__init__.py

```
from _collections import (
            deque, defaultdict )


from _collections_abc import *

class OrdererDict(dict):
    ...

class Counter(dict):
    ...
```

Controlling Exports

- Submodules should define __all__

```
# foo.py                    # bar.py

__all__ = ['Foo']           __all__ = ['Bar']

class Foo(object):          class Bar(object):
    ...                         ...
```

- Controls 'from module import *'
- Allows easy combination in __init__.py

```
# __init__.py
from .foo import *
from .bar import *

__all__ = [ *foo.__all__, *bar.__all__ ]
```

Module Splitting

- Suppose you have a large module

```
# spam.py

class Foo(object):
    ...
    ...


class Bar(object):
    ...
    ...
```

- You want to split it into multiple files
- But keep it as a single import

Module Splitting

- Step 1: Turn into a directory with multiple files

```
spam/
     foo.py
     bar.py
```

```
# foo.py

class Foo(object):
    ...
    ...
```

```
# bar.py

class Bar(object):
    ...
    ...
```

- Split the code you wish

Module Splitting

- Step 2: Stitch back together in __init__.py

```
spam/
     foo.py
     bar.py
     __init__.py
```

```
# foo.py

class Foo(object):
    ...
    ...
```

```
# bar.py

class Bar(object):
    ...
    ...
```

```
# __init__.py

from .foo import Foo
from .bar import Bar
```

Exercise 9.3

20 minutes

Import Machinery

- You can interact with import implementation
- Example: importing a module by name

```
from importlib import import_module

# import name
mod = import_module('name')

# from . import name
mod = import_module('name', __package__)
```

- Potential use: Dynamic imports or imports from settings in a config file
- There is other useful functionality in importlib

Exercise 9.4

10 minutes

Main Modules

- python -m *module*
- Runs a specified module as a main program

```
spam/
     __init__.py
     foo.py
     bar.py

bash % python3 -m spam.foo      # Runs spam.foo as main
```

- Can use to enclose supporting scripts/applications within a package

Main Entry Point

- __main__.py designates an entry point
- Makes a package directory executable

```
spam/
     __init__.py
     __main__.py                 # Starting module
     foo.py
     bar.py

bash % python3 -m spam           # Run package as main
```

- More useful than you might think

Executable Subpackages

- Example

```
spam/
    __init__.py
    foo.py
    bar.py
    test/
         __init__.py
         __main__.py ───→  bash % python3 -m spam.test
         foo.py
         bar.py
```

- Could have a variety of such tools/utilities embedded within a package
- Nice feature: they stay with the package

Exercise 9.5

15 minutes

Preparing For Distribution

- Now, suppose you want to give code to others
- To do this, you create a top-level project directory that includes everything

Other support files

Source code (a python package hierarchy)

```
packagename/
    README.txt
    LICENSE.txt
    Doc/
    packagename/
           __init__.py
           foo.py
           bar.py
           utils/
                 __init__.py
                 spam.py
                 grok.py
           ...
```

The setup.py File

- Next, write a setup.py file using distutils

```
# setup.py
from distutils.core import setup

setup(name='packagename',
      version='1.0',
      author='Your Name',
      author_email='you@somemail.com',
      url='http://www.you.com/packagename',
      packages=['packagename', 'packagename.utils' ]
)
```

- Minimally includes the name of the package, version number, and a list of all package folders,

Where to put setup.py?

- setup.py goes in the top-level directory

```
             packagename/
                  README.txt
                  LICENSE.txt
 ──────────→      setup.py
                  Doc/
                      ...
                  packagename/
                         __init__.py
                         foo.py
                         bar.py
                         utils/
                               __init__.py
                               spam.py
                               grok.py
                         ...
```

MANIFEST.in

- If you have additional directories and files such as documentation, examples, etc. you should also write a MANIFEST.in file

```
include *.txt
recursive-include examples *
recursive-include Doc *
...
```

- Here, you have to identify everything that's not part of the actual source code here

Source Distributions

- Once you have setup.py, your code is now ready to distribute

```
bash % python setup.py sdist
```

- On UNIX, this creates a file such as

```
dist/packagename-1.0.tar.gz
```

- On Windows, a zip file is created

```
dist/packagename-1.0.zip
```

Installing a Package

- Users install your package via pip

```
bash % python3 -m pip install packagename-1.0.tar.gz
```

- This will install the package in the system-wide site-packages directory

```
/usr/local/lib/python3.5/site-packages
```

- Might require admin/root permission
- Alternative: Use a virtual environment

User Local Installs

- Users can optionally install a package into their own personal directory

```
bash % python3 -m pip install --user packagename-1.0.tar.gz
```

- Installs code into a location such as this

```
~/.local/lib/python3.5/site-packages
```

- Not a bad option if you are just installing packages for yourself

Exercise 9.6

10 minutes