Chapter 10

Pythonic Idioms

Chapter 10 Objectives

- **Learn some idiomatic Python expressions**
- **Define lambda functions**
- **Pack and unpack function parameters**
- **Transform lists with list comprehensions**
- **Develop and use generators**

Pythonic Idioms

A common neologism in the Python community is *Pythonic*, which can have a wide range of meanings related to program style. To say that a piece of code is Pythonic is to say that it uses Python idioms well, that it is natural or shows fluency in the language. Likewise, to say of an interface or language feature that it is Pythonic is to say that it works well with Python idioms, that its use meshes well with the rest of the language.

In contrast, a mark of *unpythonic* code is that it attempts to "write C++ (or Lisp, or Perl) code in Python"—that is, provides a rough transcription rather than an idiomatic translation of forms from another language. The concept of pythonicity is tightly bound to Python's minimalist philosophy of readability and avoiding the "there's more than one way to do it" approach. Unreadable code or incomprehensible idioms are unpythonic.

Users and admirers of Python -— most especially those considered knowledgeable or experienced -— are often referred to as *Pythonists*, *Pythonistas*, and *Pythoneers*.

The prefix *Py* can be used to show that something is related to Python. Examples of the use of this prefix in names of Python applications or libraries include Pygame, a binding of SDL to Python (commonly used to create games); PyS60, an implementation for the Symbian Series 60 Operating System; PyQt and PyGTK, which bind Qt and GTK, respectively, to Python; and PyPy, a Python implementation written in Python. The prefix is also used outside of naming software packages: the major Python conference is named PyCon.

An important goal of the Python developers is making Python fun to use. This is reflected in the origin of the name (based on the television series *Monty Python's Flying Circus*), in the common practice of using Monty Python references in example code, and in an occasionally playful approach to tutorials and reference materials.[60] For example, the metasyntactic variables often used in Python literature are *spam* and *eggs*, instead of the traditional *foo* and *bar*.

-- from the Wikipedia article on Python

The Zen of Python

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!

This was written by Tim Peters, a longtime Pythoneer. The text is printed out as a sort of Easter Egg when you execute the code "import this".

Common Python Idioms

Tuple without parens

```
names = "John","Terry","Eric"
```

Swapping two values

```
b, a = a, b
```

Tuple with one value

```
names = "John",
```

Join strings instead of using +

```
newstring = "".join("part1","part2","part3")
```

Use in where possible

```
for line in the_file:
    # parse line

for key,value in the_dictionary.iteritems():
    # use key
```

Use sorted() to get a sorted copy of a list

```
names2 = sorted(names)
```

Use dict.get() method to count items using a dictionary

```
count_of = {}
for item in list_of_items:
    count_of[item] = count_of.get(item,0) + 1
```

Initialize a list of words (like Perl's qw() operator)

```
fruits = "apple banana mango peach guava".split()
```

Get index when iterating

```
for (i,item) in enumerate(iterable):
     # i is zero-based index, item is value
```

Test value in range

```
5 < x < 10
```

Packing and unpacking

- **Use * or ****
- **Convert list or tuple to individual items**
- **Convert dict to named parameters**

There are times when a function takes any number of parameters, and you have a list or a tuple containing the values you'd like to pass in. To solve this, you can *unpack* a list or a tuple into its individual values using the * operator.

Similarly, if a function has named parameters, you can unpack a dictionary with **.

Example

```
values = [ 5, 46, 9, 3, 14 ]
# foo is a function that takes any number of integers
foo(*values)
```

Lambda functions

- **Short cut function definition**
- **Useful for functions only used in one place**
- **Frequently passed as parameter to other functions**
- **Function body is an expression; it cannot contain other code**

A lambda function is a brief function definition that makes it easy to create a function on the fly. This can be useful for passing functions into other functions, to be called later. Functions passed in this way are referred to as "callbacks". Normal functions can be callbacks as well. The advantage of a lambda function is solely the programmer's convenience. There is no speed or other advantage.

One important use of lambda functions is for key-calculation functions for the **list.sort()** method and the **sorted()** builtin function. Another is to provide event handlers in GUI programming.

The basic syntax for creating a lambda function is

```
function-name = lambda param-list: expr
```

where *param-list* is a list of function parameters, and *expr* is an expression involving the parameters. The expression is the return value of the function.

A lambda function could also be defined in the normal manner

```
def function-name(param-list):
        return expr
```

But it is not possible to use the normal syntax as a function parameter, or as an element in a list.

Example

lambda_ex.py

```
fruits =
['watermelon','Apple','Mango','KIWI','apricot','LEMON','guava']
sfruits = sorted(fruits,key=lambda e: e.lower())
print " ".join(sfruits)
```

lambda_ex.py

```
Apple apricot guava KIWI LEMON Mango watermelon
```

List comprehensions

- **Shortcut for a for loop**
- **Can be nested**

A list comprehension is a Python idiom that creates a shortcut for a for loop.

A loop like this:

```
results = []
for var in sequence:
    results.append(expr)    # where expr involves var
```

can be rewritten as

```
results = [ expr for var in sequence ]
```

A conditional if may be optionally added:

```
results = [ expr for var in sequence if expr ]
```

Examples

listcomp.py

```
fruits =
['watermelon','apple','mango','kiwi','apricot','lemon','guava']

values = [ 2, 42, 18, 92, "boom", ['a','b','c'] ]

ufruits = [ fruit.upper() for fruit in fruits ]

afruits = [ fruit for fruit in fruits if fruit.startswith('a') ]

doubles = [ v * 2 for v in values ]

print "ufruits:"," ".join(ufruits)
print "afruits:"," ".join(afruits)

print "doubles:",
for d in doubles:
    print d,
print
```

```
listcomp.py
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: apple apricot
doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']
```

Generators vs. iterators

- **Iterator is an expression that can be looped through with foreach**
- **Generator is a method that creates an iterator**
- **Generator functions use yield rather than return**

in is one of the most powerful operators in Python. It it used with **for** for looping through a set of values. The set of values is provided by an *iterator*. Python has many builtin iterators – a file object, for instance, which allows iterating through the lines in a file.

An iterator does not make a list of all its values in memory – it creates them one at a time as needed, and feeds them to the **for-in** loop. This is of course good, because it saves memory.

A function that serves as an iterator is called a generator. A generator is like a normal function, but instead of a **return** statement, it has a **yield** statement. Each time the yield statement is reached, it provides the next value in the sequence. When there are no more values, the function calls return, and the loop stops.

Example

sieve_gen.py

```
def nextPrime(limit):
    flags = [False] * (limit+1)  # initialize flags

    for i in range(2,limit):
        if flags[i]:
            continue
        for j in range(2*i,limit+1,i):
            flags[j] = True
        yield i  # execution stops until next value is requested by
for/in loop

for p in nextPrime(100):
    print p,
```

sieve_gen.py

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
89 97
```

mary_gen.py

```
def chomper(file_obj):
    for line in file_obj:
        if line.endswith('\n'):
            line = line[:-1]
        yield line

m = open('../DATA/mary.txt')
for line in chomper(m):
    print line
m.close()
```

```
mary_gen.py
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that mary went
The lamb was sure to go
```

Generator Expressions

- **Like list comprehensions, but create a generator object**
- **More efficient**

A generator expression is similar to a list comprehension, but it provides a generator instead of a list. That is, while a list comprehension returns a complete list, the generator expression returns one item at a time.

The main difference in syntax is that the generator expression uses parentheses rather than brackets.

Generator expressions are especially useful with functions like sum(), min(), and max() that reduce an iterable input to a single value:

There is an implied **yield** statement at the beginning of the expression.

Example

mary.txt

```
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that mary went
The lamb was sure to go
```

gen_ex.py

```
#!/usr/bin/python
import re

# sum the squares of a list of numbers
# using list comprehension, entire list is stored in memory
s1 = sum([x*x for x in range(10)])
# only one square is in memory at a time with generator expression
s2 = sum(x*x for x in range(10))
print s1,s2
print

# set constructor -- does not put all words in memory
pg = open("../DATA/mary.txt")
s = set(word.lower()  for line in pg for word in re.split(r'\W+',line))
pg.close()
print s
print

# dictionary constructor
keylist = ['OWL','Badger','bushbaby','Tiger','GORILLA','AARDVARK']
d = dict( (k.lower(),k) for k in keylist)
print d
print

# find longest line in file
page = open("../DATA/mary.txt")
m = max(len(line)  for line in page if line.strip())
page.close()
print m
```

gen_ex.py

```
285 285

set(['a', 'lamb', 'little', 'sure', 'and', '', 'that', 'snow',
'went', 'the', 'had', 'fleece', 'its', 'to', 'as', 'everywhere',
'go', 'white', 'was', 'mary'])

{'owl': 'OWL', 'aardvark': 'AARDVARK', 'tiger': 'Tiger',
'gorilla': 'GORILLA', 'bushbaby': 'bushbaby', 'badger': 'Badger'}

30
```

String Tricks

• Use parentheses to spread long string over multiple lines

Because Python uses end-of-line to delimit statements, it is sometimes inconvenient to have a single string longer than 78 characters or so.

To get around this, we can take advantage of two features of Python. The first is that adjacent strings are concatenated, and the second is that statements delimited with parentheses, brackets, or braces can be spread over multiple lines.

Example
longstring.py

```
s = (
      "This is a very long string that would normally go over "
      "seventy-eight characters. In fact, this string, once "
      "concatenated, will have 301 characters.  Sometimes this "
      "could be necessary for building config files, for inserting "
      "various kinds of embedded scripts, or other interesting "
      "kinds of textual data"
)

print len( s )
print s
```

longstring.py

```
301
This is a very long string that would normally go over seventy-
eight characters. In fact, this string, once concatenated, will
have 301 characters.  Sometimes this could be necessary for
building config files, for inserting various kinds of embedded
scripts, or other interesting kinds of textual data
```

Chapter 10 Exercises

Exercise 10-1

Read all the presidents' names into a list, then use a list comprehension to print them all out in upper case.

Exercise 10-2

Print out all the presidents, date of death, and their political affiliations, sorted by date of death (or current date, if they are still living). Read the file, putting the three fields into a list of lists (or list of dictionaries). Then sort the list using LIST.sort() or sorted() and a lambda function.

Exercise 10-3

Write a generator function to provide a sequence of the names of presidents (in "FIRSTNAME MIDDLENAME LASTNAME" format) from the presidents.txt file. They should be provided in the same order they are in the file. You should not read the entire file into memory, but one-at-a-time from the file.

Chapter 11

Modules and Packages

Chapter 11 Objectives

- **Using the** import **statement to load modules**
- **Understanding .pyc files**
- **Setting locations to search for local modules**
- **Loading modules from zip files**
- **Creating local (user-defined) modules**
- **Building packages containing multiple modules**
- **Aliasing module and package names for convenience**

What is a module?

- **Sharable library of python subroutines**
- **Can have initialization code**

A module is a library of python subroutines. Any time a function might be useful in more than one place, it should be put in a module, in order to avoid cut-and-paste disease.

If the function needs to change, you only need to change it in one place.

Any code in a module that is not inside a function definition will be executed the first time a module is loaded, and thus can provide initialization code.

Modules end in **.py**.

Modules can also be written in C++ or other languages, and implemented as binary libaries (**.so** or **.DLL**).

The import statement

- **Used to load a module**
- **can import entire module, or specified functions**

The import statement is used to load modules. It can be used in several ways:

`import spam`

Load module s**pam.py**. To call function **Eggs** which is defined in s**pam.py**, use s**pam.Eggs**

```
spam.Eggs("scrambled")
```

`from spam import Eggs[,Toast,Coffee,...]`

Load module s**pam.py** and add **Eggs** to the current namespace, so it can be called without the module name:

```
Eggs("fried")
```

To import from multiple modules, put them in a comma-separated list.

`from spam import *`

Load module spam.py and add all functions (except those than begin with "_") to the current namespace.

```
Eggs("poached")
toast("butter","jam")
```

This is generally not considered a Good Thing unless you know what you're doing, or the module docs suggest it.

Though they still use the import statement, some modules are built into the Python interpreter, rather than being loaded from a file

Where did that .pyc file come from?

- import **precompiles a module**
- **Loading (but not execution) is faster**

After running a script which uses module c**heese.py** in the current directory, you will notice that a file named c**heese.pyc** has appeared. The Python interpreter saves a compiled version of each imported module into the directory containing the original module.

This speeds up the loading of modules, as whitespace, comments, and other non-code items are removed. It does not, however, speed up the execution of the module, as, once loaded, the code in memory is the same.

Subsequent invocations of the script using the module will load the .pyc version. If the modification date of the original (.py) file is later than the .pyc, the interpreter will recompile the module.

Module search path

- **Searches current dir first, then predefined locations**
- **Paths stored in sys.path**
- **Append to sys.path for user-defined locations**

When you specify a module to load with the **import** statement, it first looks in the current directory, and then searches the directories listed in **sys.path**.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python24.zip', '/usr/lib/python2.4',
'/usr/lib/python2.4/plat-linux2', '/usr/lib/python2.4/lib-
tk', '/usr/lib/python2.4/lib-dynload',
'/usr/local/lib/python2.4/site-packages',
'/usr/lib/python2.4/site-packages',
'/usr/lib/python2.4/site-packages/HTMLgen',
'/usr/lib/python2.4/site-packages/Numeric',
'/usr/lib/python2.4/site-packages/PIL',
'/usr/lib/python2.4/site-packages/cairo',
'/usr/lib/python2.4/site-packages/gtk-2.0',
'/usr/lib/python2.4/site-packages/wx-2.6-gtk2-unicode']
```

You can also put one or more directories to search in the **PYTHONPATH** environment variable, separated by semicolons for Windows, or colons for Unix/Linux. This is preferable to hard-coding directories in your scripts.

Windows

```
set PYTHONPATH=C:"\Documents and settings\Bob\Python"
```

Unix

```
export PYTHONPATH="/home/bob/python"
```

You can also append directories to sys.path before importing those modules

```
import sys
sys.path.extend("/usr/dev/python/libs","/home/bob/pylib")
import stuff
import morestuff
```

Zipped Libraries

- **Import module from zipped file rather than folder**
- **Add zip file to sys.path**

To save space, or to distribute a package, Python can load a library from a zip file. To do this, add one or more zip files to **sys.path**.

Example

```
import sys
sys.path.append("mylib.zip")
import Foo,Bar    Foo.py and Bar.py are extracted from MyLib.zip
```

Windows

```
set PYTHONPATH="C:\Documents and settings\Bob\CoolApp.zip"
```

Unix

```
export PYTHONPATH="/home/bob/coolapp.zip"
```

Creating Modules

- **No special syntax**
- **Ends in .py**
- **Add a doc string**

You can create your own modules easily. Any valid Python source file may be loaded as a module.

It is a good idea to provide a doc string for each function. This is a string which is the first statement in the body of a function. It will be used by automated documentation string; it is available via the .__doc__ attribute of the function object as well.

The naming convention is for module names to be lower case.

To test a module (i.e. for syntax errors), you can run it as a python script:

```
python module.py
```

You can put in some self-testing code by examining the builtin attribute **__name__**. If **__name__** is "**__main__**", then the module is running as a script; otherwise it has been loaded as a module:

```
if __name__ == "__main__":
        print "Testing...."
        # put testing code here
```

Example

spam.py

```
# Module Spam -- provides a tasty breakfast

# separator for toast toppings
topsep = " and "

def eggs(how):
        "cook some eggs"
        print "Cooking up some lovely {0} eggs".format(how)

def toast(*toppings):
        """cook some toast
           -- add dairy products
           -- add fruity spreads """
        print  "Toasting up some toast with ",
                   topsep.join(toppings)
```

eat.py

```
from Spam import eggs,toast

eggs("fried")
toast("butter","strawberry jam")

print
print "What does eggs() do?"
print eggs.__doc__
print
print "What does toast() do?"
print toast.__doc__
```

eat.py

```
Cooking up some lovely fried eggs
Toasting up some toast with  butter and strawberry jam

What does eggs() do?
cook some eggs

What does toast() do?
cook some toast
            -- add dairy products
            -- add fruity spreads
```

Packages

- **Collection of modules**
- **Corresponds to a directory**
- **Can have subpackages**

A package is a collection of modules that have been grouped in a directory for convenience.

Load modules in a package with `import package`

Access subpackages via `PACKAGE.SUB`

Access functions via `PACKAGE.SUB.function()`

For instance, the following directory structure implements package media, which includes modules cd, dvd, and videotape:

media

media/__init__.py

media/dvd.py

media/videotape.py

media/cd.py

To use the function Search that is defined in module dvd, import media.dvd:

```
import media.dvd
media.dvd.Search("Uma Thurman")
```

Other variations are:

```
from media import dvd
dvd.Search("Uma Thurman")
```

or

```
from media.dvd import Search
Search("Uma Thurman")
```

The __init__.py script is loaded (and executed) the first time any module in the Media package is loaded. It must be present, but may be empty. It is used for initializing the entire package.

Module Aliases

Provide alternate name for module

To load a module with a different name, use the syntax

```
import module as alias
```

This is useful to save typing, especially when using packages.

Example

```
import media.dvd as dvd
found = dvd.search('beatles')

import xml.etree.ElementTree as ET
import Tkinter as tk
```

When the batteries aren't included

- **Python Package Index (PyPI) has 22000 packages**

Although the Python distribution claims to be "batteries included", functionality beyond the standard library is provided by so-called third party modules. There are about 22,000 such packages in the Python Package Index (http://pypi.python.org/pypi).

These modules can be installed with the **easy_install** utilitiy.

Another installer is pip, meant to be a replacement for easy_install. It is more Unix oriented, and may not work as well as easy_install in some cases, although in general, it is easier to use.

Chapter 11 Exercises

Exercise 11-1

Create a module named **tempconv** that contains two functions, **c2f** and **f2c**. Implement the functions, and write a testing script to make sure they work.

Formulae are:

F = ((9 * C) / 5.0) + 32

C = (F - 32) * (5.0/9)

Exercise 11-2

Re-implement exercise 3-1 using the new **tempconv** module.

Exercise 11-3

Re-implement exercise 4-1 using **tempconv**.

Chapter 12

Classes

Chapter 12 Objectives

- **Understanding the big picture of OO programming**
- **Defining a class**
- **Creating the constructor**
- **Implementing object methods and properties**
- **Using inheritance for code reuse**
- **Adding class data and methods**

Defining Classes

- **Syntax**

```
class classname(base_class,...):
    # class body - methods and data
```

- **Classes create a namespace**

The **class** statement defines a class and assigns it to a name.

The simplest form of class definition looks like this:

```
class ClassName(object):
    pass
```

Normally, the contents of a class definition will be method definitions and shared data.

A class definition creates a new local namespace. All variable assignments go into this new namespace. All methods are called via the class name.

A list of base classes is specified in parentheses after the class name. Always specify at least one base class. If you do not need to inherit from a specific class, use the default base class **object.**

> Class definitions that do not specify a base class are legal, but discouraged. They create what are known as "old-style classes".

Instance Objects

- **Call class name as a function**
- ***self* contains attributes: properties & methods**
- **Syntax**

```
obj = Class(args)
```

An *object instance* is an object created from a class. Each object instance has its own private attributes, which were perhaps initialized in the **__init__** method.

Instance attributes

- **Methods and data**
- **Accessed using dot notation**
- **Privacy by convention *(_name)***

An instance of a class (AKA object) normally contains methods and data. To access these attributes, use "dot notation": *object.attribute*.

Instance attributes are dynamic; they can be accessed directly from the object. You can create, update, and delete attributes in this way.

Attributes cannot be made private, but names that begin with an underscore are understood by convention to be for internal use only. Users of your class will not consider methods that begin with an underscore to be part of your class's API. The **from *module* import *** idiom does not import attributes that begin with an underscore.

Example

```
class Spam(object):
    def eggs(self):
        pass

    def _beverage(self):   # private!
        pass

s = Spam()
s.eggs()
s.toast = 'buttered'
print s.toast

s._beverage()   # legal, but wrong!
```

In many cases, it is better to use **properties** (described later) to access data attributes.

Methods

- **Called from objects**
- **Passed invoking objects as implicit parameter**

A method is a function defined in a class. When a method is called from an object, the object is passed in as the implicit first parameter, named self by strong convention.

Example

rabbit.py

```
class Rabbit:

    def __init__(self,size,danger):
        self._size = size
        self._danger = danger
        self._victims = []

    def threaten(self):
        print "I am a %s bunny with %s!" % (self._size, self._danger)

r1 = Rabbit('large',"sharp, pointy teeth")
r1.threaten()

r2 = Rabbit('small','fluffy fur')
r2.threaten()
```

In C++, Java, and C#, ***self*** might be called ***this***.

__init__

- **Constructor**
- **Implicitly called when object is created**

If a class defines a method named **__init__** , it will be automatically called when an object instance is created. This is the *constructor.*

The object being created is implicitly passed as the first parameter to **__init__** . This parameter is named **self** by very strong convention. Data attributes can be assigned to **self.** These attributes can then be accessed by other methods.

Example

```
class Rabbit:

    def __init__(self,size,danger):
        self._size = size
        self._danger = danger
        self._victims = []
```

Properties

- **Accessed like attributes**
- **Invoke implicit getters, setters, and deleters**

While object attributes can be accessed directly, in many cases the class needs to exercise some control over the attributes.

One approach is to provide *getter, setter,* and *deleter* methods for attributes. These are methods that can be called to indirectly manage attributes. There are many variations on the implementation of these methods.

Example

```
class Knight(object):
    def __init__(self,name):
        self._name = name

    def set_name(self,name):
        self._name = name

    def get_name(self):
        return self._name

k = Knight("Lancelot")
print k.get_name()
```

A more elegant approach is to use *properties*. A property is sometimes called a *managed attribute*. Properties are accessed directly, like normal attributes, but getter, setter, and deleter functions are implicitly called, so that the class can control what values are stored or retrieved from the attributes.

There are two ways to create properties. Starting in Python 2.2, the **property()** function could be used to create a property by specifying a getter, setter (optional), and deleter (optional) method. Starting in 2.6, an existing property has attributes named **getter, setter,** and **deleter**, which can be used to implement the corresponding functionality.

Example (>= 2.6)

knight_managed.py

```
class Knight(object):
    def __init__(self,name):
        self._name = name

    @property
    def Name(self):
        return self._name

    @Name.setter
    def Name(self,name):
        self._name = name

    @property
    def Color(self):
        return self._color

    @Color.setter
    def Color(self,color):
        self._color = color

k = Knight("Lancelot")
k.Color = 'blue'

# Bridgekeeper's question
print 'Sir %s, what is your...favorite color?' % ( k.Name )
# Knight's answer
print "red, no -- %s!" % ( k.Color )
```

knight_managed.py

```
Sir Lancelot, what is your...favorite color?
red, no - blue!
```

Example (< 2.6)

knight_managed_old.py

```
class Knight(object):
    def __init__(self,name):
        self._name = name

    @property
    def Name(self):  # simple read-only property
        return self._name

    # read-write property prior to Python 2.6
    def _getcolor(self):
        return self._color

    def _setcolor(self,color):
            self._color = color

    Color = property(_getcolor,_setcolor,None,'Favorite Color')

k = Knight("Lancelot")
k.Color = 'blue'

# Bridgekeeper's question
print 'Sir %s, what is your...favorite color?' % ( k.Name )
# Knight's answer
print "red, no -- %s!" % ( k.Color )
```

```
knight_managed_old.py
Sir Lancelot, what is your...favorite color?
red, no – blue!
```

Class Data

- **Attached to class, not instance**
- **Shared by all instances**
- **Use @classmethod to create class methods**

Data can be attached to the class itself, and shared among all instances. Class data can be accessed via the class name from inside or outside of the class.

If a method only needs class attributes, it can be made a class method via the **@classmethod** decorator. This alters the method so that it gets a copy of the class object rather than the instance object. The parameter to a class method is named **cls** by strong convention.

> Any class attribute not overwritten by an instance attribute is also available through the instance.

Example

classmethod.py

```
class Rabbit:
    weapon = 'pointy teeth'

    def get_weapon(self):
        print "INSTANCE: (%s) %s" % (self,Rabbit.weapon)

    @classmethod
    def get_weapon_again(cls):
        print "CLASS: (%s) %s" % (cls,cls.weapon)

r = Rabbit()

r.get_weapon()
r.get_weapon_again()
Rabbit.get_weapon_again()
print Rabbit.weapon
```

classmethod.py

```
INSTANCE: (<__main__.Rabbit instance at 0xb77dc52c>) pointy teeth
CLASS: (__main__.Rabbit) pointy teeth
CLASS: (__main__.Rabbit) pointy teeth
pointy teeth
```

Inheritance

- **Specify base class in class definition**
- **Call base class constructor explicitly**

Of course, a language feature would not be worthy of the name "class"' without supporting inheritance. One or more base classes may be specified as part of the class definition. All of the previous examples in this chapter have used the default base class, **object.**

The base class must already be imported, if necessary. If a requested attribute is not found in the class, the search looks in the base class. This rule is applied recursively if the base class itself is derived from some other class.

Classes may override methods of their base classes. (For Java and C++ programmers: all methods in Python are effectively `virtual`.)

To extend rather than simply replace a base class method, call the base class method directly: `BaseClassName.methodname(self, arguments).`

Example

confuse.py

```
class Confuse(object):

    def __init__(self,target,action):
        self.target = target
        self.action = action

    def stun(self):
        if self.action.endswith("e"):
            suffix = "rs"
        elif self.action[-1] in "bdfglmnprst":
            suffix = self.action[-1] + "ers"
        else:
            suffix = "ers"

        t =  (self.target.capitalize(),self.action,suffix)
        print "Squad! Eyes front! Stand at ease. %s %s%s ...shun!" % t
```

confuse_a_cat.py

```
from confuse import Confuse

class ConfuseACat(Confuse):
    def __init__(self):
        Confuse.__init__(self,"cat","confuse")
```

amaze_a_vole.py

```
from confuse import Confuse

class AmazeAVole(Confuse):
    def __init__(self):
        Confuse.__init__(self,"vole","amaze")
```

stun_a_stoat.py

```
from confuse import Confuse

class StunAStoat(Confuse):
    def __init__(self):
        self.target = "stoat"
        self.action = "stun"
```

confusion.py

```
from confuse_a_cat import ConfuseACat
from amaze_a_vole import AmazeAVole
from stun_a_stoat import StunAStoat

c = ConfuseACat()
c.stun()

a = AmazeAVole()
a.stun()

s = StunAStoat()
s.stun()
```

confusion.py

```
Squad! Eyes front! Stand at ease. Cat confusers ...shun!
Squad! Eyes front! Stand at ease. Vole amazers ...shun!
Squad! Eyes front! Stand at ease. Stoat stunners ...shun!
```

Multiple Inheritance

- **More than one base class**
- **All data and methods are inherited**
- **Methods resolved left-to-right, depth-first**

Python classes can inherit from more than one base class. This is called "multiple inheritance". A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The method search algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents).

For more information on method resolution order (MRO), see http://www.python.org/download/releases/2.3/mro/.

Base Classes

- **Also called abstract or virtual**
- **Define methods needed by derived classes**
- **Virtual methods raise NotImplementedError**

Python doesn't officially support the notion of an abstract or virtual class – that is, a class that *must* be subclassed and cannot be directly instantiated.

However, you can provide base classes that are something like this by using **NotImplementedError.** To create "virtual" methods, raise **NotImplementedError** in the body of the method. If these methods are not overwritten in the derived class, they will raise **NotImplementedError** when they are called.

Example

baseclasses.py

```
class BaseClass(object):

    # non-virtual method
    def MethodA(self):
        print "In BaseClass: Hello from MethodA"

    # virtual method
    def MethodB(self):
        raise NotImplementedError

class ChildClass(BaseClass):

        def MethodB(self):
            print "In ChildClass: Hello from MethodB"

if __name__ == '__main__':

    b = BaseClass();
    b.MethodA()
    try:
        b.MethodB()
    except NotImplementedError, e:
        print "MethodB not implemented in BaseClass"

    c = ChildClass()
    c.MethodA()
    c.MethodB()
```

baseclasses.py

```
In BaseClass: Hello from MethodA
MethodB not implemented in BaseClass
In BaseClass: Hello from MethodA
In ChildClass: Hello from MethodB
```

Special Methods

- **User-defined classes emulate standard types**
- **Define behavior for builtin functions**
- **Override operators**

Python has a set of special methods that can be used to make user-defined classes emulate the behavior of builtin classes. These methods can be used to define the behavior for builtin functions such as **str()**, **len()** and **repr()**; they can also be used to override many Python operators, such as **+**, *****, and **==**.

These methods expect the ***self*** parameter, like all instance methods. They frequently take one or more additional methods. ***self***. Is the object being called from the builtin function, or the *left* operand of a binary operator such as **==**.

For instance, if your object represented a database connection, you could have str() return the hostname, port, and maybe the connection string. The default for str() is to call repr(), which returns something like **<__main__.DBConn object at 0xb7828c6c>**, which is not nearly so user-friendly.

See http://docs.python.org/reference/datamodel.html#special-method-names for detailed documentation on the special methods.

Special Methods and Variables	
Method or Variables	**Description**
__new__(cls,...)	Returns new object instance; Called before __init__()
__init__(self,...)	Object initializer (constructor)
__del__(self)	Called when object is about to be destroyed
__repr__(self)	Called by repr() builtin
__str__(self)	Called by str() builtin
__eq__(self, *other)* __ne__(self, *other*) __gt__(self, *other*) __lt__(self, *other*) __ge__(self, *other*) __le__(self, *other*)	Implement comparison operators ==, !=, >, <, >=, and <=. self is object on the left.
__cmp__(self, *other*)	Called by comparison operators if __eq__, etc., are not defined
__hash__(self)	Called by hash() builtin, also used by dict, set, and frozenset operations
__nonzero__(self)	Called by bool() builtin. Implements truth value (boolean) testing. If not present, bool() uses len()
__unicode__(self)	Called by unicode() builtin
__getattr__(self, *name)* __setattr__(self, *name*, *value)* __delattr__(self, *name*)	Override normal fetch, store, and deleter
__getattribute__(self, *name*)	Implement attribute access for new-style classes
__get__(self, *instance*) __set__(self, instance, value) __del__(self, *instance*)	Implement descriptors
__slots__ = *variable-list*	Allocate space for a fixed number of attributes.
__metaclass__ = *callable*	Called instead of **type()** when class is created.
__instancecheck__(self, *instance)*	Return true if *instance* is an instance of class
__subclasscheck__(self, *instance)*	Return true if *instance* is a subclass of class
__call__(self, ...)	Called when instance is called as a function.
__len__(self)	Called by len() builtin
__getitem__(self, *key)*	Implements self[*key*]
__setitem__(self, *key*, *value*)	Implements self[*key*] = *value*
__selitem__(self, *key*)	Implements del self[*key*]
__iter__(self)	Called when iterator is applied to container
__reversed__(self)	Called by reversed() builtin
__contains__(self, *object)*	Implements **in** operator

Special Methods and Variables

Method or Variables	Description
__add__(self, *other)* __sub__(self, *other*) __mul__(self, *other)* __floordiv__(self, *other*) __mod__(self, *other)* __divmod__(self, *other)* __pow__(self, *other[, modulo])* __lshift__(self, *other)* __rshift__(self, *other*) __and__(self, *other)* __xor__(self, *other*) __or__(self, *other)*	Implement binary arithmetic operators +, -, *, //, %, **, <<, >>, &, ^, and |. Self is object on left side of expression.
__div__(self,other) *__truediv__(self,other)*	Implement binary division operator /. **__truediv__** is called if **__future__.division** is in effect.
__radd__(self, *other)* __rsub__(self, *other*) __rmul__(self, *other*) __rdiv__(self, *other*) __rtruediv__(self, *other)* __rfloordiv__(self, *other*) __rmod__(self, *other)* __rdivmod__(self, *other)* __rpow__(self, *other)* __rlshift__(self, *other)* __rrshift__(self, *other*) __rand__(self, *other*) __rxor__(self, *other*) __ror__(self, *other)*	Implement binary arithmetic operators with swapped operands. (Used if left operand does not support the corresponding operation)

Special Methods and Variables	
Method or Variables	**Description**
__iadd__(self, *other)* __isub__(self, *other*) __imul__(self, *other)* __idiv__(self, *other*) __itruediv__(self, *other*) __ifloordiv__(self, *other*) __imod__(self, *other)* __ipow__(self, *other[, modulo])* __ilshift__(self, *other*) __irshift__(self, *other*) __iand__(self, *other*) __ixor__(self, *other*) __ior__(self, *other*)	Implement augmented (+=, -=, etc.) arithmetic operators.
__neg__(self) __pos__(self) __abs__(self) __invert__(self)	Implement unary arithmetic operators **-**, **+**, **abs()**, and **~**.
__oct__(self) __hex__(self)	Implement **oct()** and **hex()** builtins
__index__(self)	Implement **operator.index()**
__coerce__(self, *other*)	Implement "mixed-mode" numeric arithmetic.

Example

specialmethods.py

```
class Special(object):

    def __init__(self,value):
        self._value = value

    # define ... Special object ... added to another Special object
    def __add__(self,other):
        return self._value + other._value

    # define what happens when a Special object is multiplied by a
value
    def __mul__(self,num):
        return ''.join((self._value for i in range(num)))

    # define what happens when str() called on a Special object
    def __str__(self):
        return self._value.upper()

    def __eq__(self,other):
        return self._value == other._value

if __name__ == '__main__':
    s = Special('spam')
    t = Special('eggs')
    u = Special('spam')

    print "s + s", s + s
    print "s + t", s + t
    print "t + t", t + t
    print "s * 10", s * 10
    print "t * 3", t * 3
    print "str(s)=%s  str(t)=%s" % ( str(s), str(t) )
    print "id(s)=%d id(t)=%d id(u)=%d" % ( id(s), id(t), id(u) )
    print "s == s", s == s
    print "s == t", s == t
    print "u == s", s == u
```

specialmethods.py

```
s + s spamspam
s + t spameggs
t + t eggseggs
s * 10 spamspamspamspamspamspamspamspamspamspam
t * 3 eggseggseggs
str(s)=SPAM  str(t)=EGGS
id(s)=3078139180 id(t)=3078139212 id(u)=3078139244
s == s True
s == t False
u == s True
```

Pseudo-Private Variables

- **Provide variables private to derived classes**
- **Start with 2 underscores**
- **At most 1 trailing underscore**

Python supports class-private variables in a hackish way. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped.

This mangling is done for all identifiers in a class. Outside classes, or when the class name consists of only underscores, no mangling occurs.

Thus, a base and derived class can both have the attribute **customer_num** without one overwriting the other.

These mangling rules are designed mostly to avoid accidents; it still is possible for a determined programmer to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger, and that's one reason why this loophole is not closed.

Truncation may occur when the mangled name would be longer than 255 characters.

Static Methods

- **Related to class, but doesn't need instance or class object**
- **Use *@staticmethod* decorator**

A *static method* is a utility method that is functionally related to the class, but does not need the instance or class object. Thus, it has no automatic parameter. Static methods are seldom needed.

One use case for static methods is when a class wants to provide a convenience method.

Example

clinic.py

```
class ArgumentClinic(object):
    @staticmethod
    def argue():
        return "No it isn't."
```

argue.py

```
from clinic import ArgumentClinic

a = ArgumentClinic()

for i in xrange(1,25):
    print a.argue(),
```

argue.py

```
No it isn't. No it isn't. No it isn't. No it isn't. No it isn't. No it isn't. No
 it isn't. No it isn't. No it isn't. No it isn't. No it isn't. No it isn't. No i
t isn't. No it isn't. No it isn't. No it isn't. No it isn't. No it isn't. No it
isn't. No it isn't. No it isn't. No it isn't. No it isn't. No it isn't.
```

Chapter 12 Exercises

Exercise 12-1

Create a module named **president** that implements a President class. This class has a constructor that takes the index number of the president (1-44) and creates an object containing the associated information from the presidents.txt file.

Provide the following properties (types indicated after -->):

```
TermNumber --> int

Affiliation --> string

BirthPlace --> string

TermStartDate --> date object

TermEndDate --> date object (or None, if still in office)

BirthDate --> date object

DeathDate --> date object (or None, if still alive)
```

Write a test script to exercise all of the methods. It could look something like

```
from president import President

months = ["","Jan","Feb","Mar","Apr","May","Jun",
   "Jul","Aug","Sep","Oct","Nov","Dec" ]

p = President(1)   # George Washington

bd = p.BirthDate

print "George was born on %s %d, %d" %
(months[bd.month],bd.day,bd.year)
```

For the ambitious: Revise the constructor so it takes either an index or a name n the form "Last, First M.".

Chapter 13

Developer Tools

Chapter 13 Objectives

- **Run pylint to check source code**
- **Create basic unit test cases**
- **Run test cases**
- **Skip tests with decorators**
- **Discover test cases automatically**
- **Debug scripts**
- **Find speed bottlenecks in code**
- **Compare algorithms to see which is faster**

Program development

- **More than just coding**
  - **-- Design first**
  - **-- Consistent style**
  - **-- Comments**
  - **-- Debugging**
  - **– Testing**
  - **-- Documentation**

Comments

- **Keep comments up-to-date**
- **Use complete sentences**
- **Block comments describe a section of code**
- **Inline comments describe a line**
- **Don't state the obvious**

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period. Use two spaces after a sentence-ending period.

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement; they should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1        # Increment x
```

Only use an inline comment if the reason for the statement is not obvious.

```
x = x + 1        # Compensate for border
```

The above was adapted from PEP 8.

See Pep 257 for detailed suggestions for doc strings

pylint

- **Checks many aspects of code**
- **Finds mistakes**
- **Rates your code for standards compliance**
- **Don't worry if your code has a lower rating!**
- **Can be highly customized**

pylint *is a Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells (as defined in Martin Fowler's Refactoring book)*

from the pylint documentation

pylint can be very helpful in identifying errors and pointing out where your code does not follow standard coding conventions.

It has very verbose output, which can be modified via command line options.

pylint can be customized to reflect local coding conventions.

To use **pylint**, just say **`pylint filename`**, or **`pylint folder`**.

Other tools for analyzing Python code:

pyflakes

pychecker

Customizing pylint

- **Use pylint –generate-rcfile**
- **Redirect to file**
- **Edit as needed**
- **Knowledge of regular expressions useful**
- **Name file ~/.pylintrc on Linux/Unix/OS X**
- **Use –rcfile *file* to specify custom file on Windows**

To customize **pylint**, run **pylint** with only the **-generate-rcfile** option. This will output a well-commented configuration file to STDOUT, so redirect it to a convenient location.

Edit the file as needed. The comments describe what each part does. You can change the allowed names of variables, functions, classes, and pretty much everything else. You can even change the rating algorithm.

Windows

Put the file in a convenient location (name it something like **pylintrc**). Invoke **pylint** with the **–rcfile** option to specify the location of the file.

pylint will also find a file named **pylintrc** in the current directory, without needing the **-rcfile** option.

Non-Windows systems

On Unix-like systems (Unix, Mac OS, Linux, etc.), **/etc/pylintrc** and **~/.pylintrc** will be automatically loaded, in that order.

See docs.pylint.org for more details.

Using pyreverse

- **Source analyzer**
- **Reverse engineers Python code**
- **Part of pylint**
- **Generates UML diagrams**

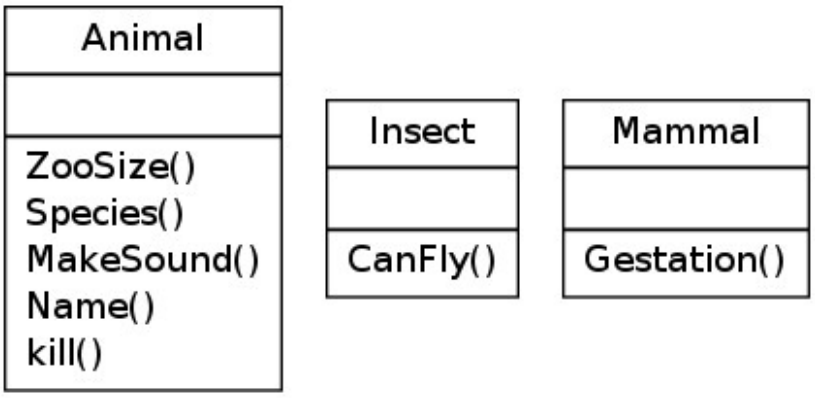
pyreverse is a Python source code analyzer. It reads a script, and the modules it depends on, and generates UML diagrams. It is installed as part of the **pylint** package.

There are many options to control what it analyzes and what kind of output it produces.

Example

```
pyreverse -o png -p animal -A animal.py mammal.py insect.py
```

classes_class_tree.png



The unittest Module

- **Provides automation for testing**
- **Test classes inherit from** `unittest.TestCase`

A *unit test* is a test which asserts that an isolated piece of code (one function, method, or class) has some expected behavior. It is a way of making sure that code provides repeatable results.

The `unittest` module provides base classes and tools for creating, running, and managing unit tests.

There are three main components of a unit testing system:

1. Unit tests – individual assertions that an expected condition has been met

2. Test cases – collections of related unit tests

3. Test runners – utilities to execute the tests in one or more test cases

Unit tests should each test one aspect of your code, and each test should be independent of all other tests, including the order in which tests are run.

Unit tests are collected into a *test case*, which is a related group of unit tests. You can create test cases by inheriting from **unittest.TestCase**.

Individual test names must begin with "test", so they can be discovered by automated test runners. It is conventional to make test names verbose, so when test names are output, it is clear which tests are being run.

Each test makes an *assertion*; that is, it asserts that some condition is true. In addition to the builtin **assert** function, unittest.TestCase provides many specialized assertion functions to give better reporting when a test fails.

Another component of a unit test system is a *test suite*, which is a collection of test cases or other test suites. **unittest** also provides test suite builders.

unittest Assertions	
Methods	**Assert that...**
assertAlmostEqual assertNotAlmostEqual	Two expressions are equal [unequal] as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the difference between the two objects is more than the given delta.
assertDictContainsSubset	First dictionary is a superset of the second.
assertDictEqual	Two dictionaries have the same keys and values
assertEqual assertNotEqual	Two expressions are equal [unequal] as determined by the '==' operator.
assertGreater	The first expression is greater than the second
assertGreaterEqual	The first expression is greater than or equal to the second
assertIn assertNotIn	The first expression is [not] a member of the second
assertIs assertIsNot	The first expression is [not] the same object as the second
assertIsInstance assertNotIsInstance	The first expression is [not] an instance of the second
assertIsNone assertIsNotNone	The expression is [not] None
assertItemsEqual	The first expression and second expression have the same element counts (and the same elements, but not necessarily in the same order)
assertLess	The first expression is less than the second
assertLessEqual	The first expression is less than or equal to the second
assertListEqual	The first list is equal to the second
assertMultiLineEqual	The first multi-line strings is equal to the second
assertRaises	The specified exception is raised when the specified callable is invoked
assertRegexpMatches assertNotRegexpMatches	The expression matches [does not match] the specified regular expression (can be string or **re** instance)
assertSequenceEqual	The first ordered sequence (lists or tuples) is equal to the second
assertSetEqual	The first set is equal to the second
AssertTrue, assert_ assertFalse	The expression is True [False]
assertTupleEqual	The first tuple is equal to the second
Note: shaded methods can have a trailing 's' – assertEqual and assertEquals are equivalent	

Fixtures

- **Run before and after each test**
- **Factor out common code**
- **Predefined names: setUp() and tearDown()**
- **Class-level fixtures: setUpClass and tearDownClass()**

To avoid duplicating code across many tests, unittest provides *fixtures*. These are methods that are called before or after each individual test. They can be used for some common task, such as initializing an array or creating a class instance.

Instance methods setUp() and tearDown() are called before and after each test, if present. If you do not implement them, the default versions do nothing.

You can also implement class-level fixtures. SetUpClass() and tearDownClass() are called at the beginning and end of the entire test case.

Example

testrandom.py

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        print "Starting all tests"

    def setUp(self):
        print "Starting Test..."
        self.seq = range(10)

    def testshuffle(self):
        # make sure the shuffled sequence does not lose any elements
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

    def testchoice(self):
        element = random.choice(self.seq)
        self.assert_(element in self.seq)

    def testsample(self):
        self.assertRaises(ValueError, random.sample, self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assert_(element in self.seq)

    def tearDown(self):
        print "...Ending test."

    @classmethod
    def tearDownClass(cls):
        print "Ending all tests"

if __name__ == '__main__':
    unittest.main()
```

testrandom.py

```
Starting all tests
Starting Test...
...Ending test.
.Starting Test...
...Ending test.
.Starting Test...
...Ending test.
.Ending all tests

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Skipping tests

- **Skip tests depending on circumstances**
- **Decorate with**
  @unittest.skip(reason)
  @unittest.skipIf(true-condition, reason)
  @unittest.skipUnless(false-condition, reason)

To skip tests conditionally (or unconditionally), **unittest** provides three decorators. **unittest.skip()** skips a test unconditionally. **unittest.skipIf()** skips a test if the condition is true, and **unittest.skipUnless()** skips a test if the condition is false. The first parameter to all three decorators is a string with the reason for skipping the test.

You can also create your own decorator by creating a function that returns unittest.skip() if the test should be skipped, and the function itself otherwise.

Making a suite of tests

- **Make suites from one or more test classes**
- **Make one toplevel suite to contain all other suites**
- **Suites can be nested**
- **Create a test runner**
- **Run the toplevel suite**

Most of the time you will not be running one test class at a time. You'll want to run all the tests for a module, package, framework, or application.

To do this, use **unittest.makeSuite()** to make a suite of tests from each test class. Then use **unittest.testSuite()** to make a toplevel suite that will contain all the individual suites. Add each individual test suite to the toplevel suite.

Next, create a test runner. The default runner, included with unittest, is **unittest.TextTestRunner**. Other test runners are available from PyPI.

Pass the toplevel suite to the **run()** method of the test runner object to execute all the tests in the toplevel suite.

> In real life, you will usually use **unittest discover** or **nose** to automate test suite creation.

Example

testspam.py

```
import unittest

class TestSpam(unittest.TestCase):

    def test_one_is_one(self):
        self.assertEqual(1,1)

if __name__ == '__main__':
    unittest.main()
```

testeggs.py

```
import unittest


class TestEggs(unittest.TestCase):


    def test_two_plus_two_is_four(self):
        self.assertEqual(2 + 2, 4)


if __name__ == '__main__':
    unittest.main()
```

testrunner.py

```
import unittest
from testspam import TestSpam
from testeggs import TestEggs

# create an empty test suite
testall_suite = unittest.TestSuite()

# create test suites from TestSpam and TestEggs
spam_suite = unittest.makeSuite(TestSpam)
eggs_suite = unittest.makeSuite(TestEggs)

# add suites from TestSpam and TestEggs to the testall suite
testall_suite.addTest(spam_suite)
testall_suite.addTest(eggs_suite)

# create a generic test runner whose output goes to the screen
runner = unittest.TextTestRunner()

# run the suite of suites
runner.run(testall_suite)
```

testrunner.py

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Automated test discovery

- **Find tests (classes inheriting from unittest.TestCase)**
- **Create suites**
- **Run all tests**

The previous topic showed how to create suites of tests to run all the tests in a project. Because this is both tedious and straightforward, Python 2.7 (and 3.2) added tools to automate finding running tests.

While you can use it programmatically, the usual approach is to run the **discover** method of **unittest** from the command line as follows:

```
python -m unittest discover
```

The following options can be added:

```
-v, --verbose           verbose output
-s, --start-directory   starting directory (defaults to .)
-p, --pattern           pattern for test modules (defaults to test*.py)
-t, --top-level-directory       top level of project (defaults to start directory)
```

This will find and run all tests in the specified folder and subfolders.

Using Nose

- **Discovers (sniffs out) tests**
- **Just run nosetests**
- **No need to create suites**
- **Many tools based on Nose**

Nose is a very popular test discovery tool. There are many third-party utilities that are based on Nose.

Once Nose is installed, go to the folder containing the test classes and execute the **nosetests** command. It will find all the tests and run them with the **TextTestRunner**. **nosetests** has a large number of options to fine-tune its operation.

The Python debugger

- **Implemented via pdb module**
- **Supports breakpoints and single stepping**
- **Based on** gdb

While most IDEs have an integrated debugger, it is good to know how to debug from the command line. The **pdb** module provides debugging facilities for Python.

The usual way to use **pdb** is from the command line:

```
python -mpdb script_to_be_debugged.py
```

Once the program starts, it will pause at the first executable line of code and provide a prompt, similar to the interactive Python prompt. There is a large set of debugging commands you can enter at the prompt to step through your program, set breakpoints, and display the values of variables.

Since you are in the Python interpreter as well, you can enter any valid Python expression.

You can also start debugging mode from within a program.

Starting debug mode

- **Syntax**

```
python -m pdb script
```

or

```
import pdb
pdb.run('function')
```

pdb is usually invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import some_module
>>> pdb.run('some_module.function_to_text()')
> <string>(0)?()
(Pdb) c    # (c)ontinue
> <string>(1)?()
(Pdb) c    # (c)ontinue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

To get help, type **h** at the debugger prompt.

Stepping through a program

- **s** *single-step, stepping into functions*
- **n** *single-step, stepping over functions*
- **r** *return from function*
- **c** *run to next breakpoint or end*

The debugger provides several commands for stepping through a program. Use **s** to step through one line at a time, stepping into functions.

Use **n** to step over functions; use **r** to return from a function; use **c** to continue to next breakpoint or end of program.

Pressing **Enter** repeats most commands; if the previous command was list, the debugger lists the *next* set of lines.

Setting breakpoints

- **Syntax**

```
b                                   list all breaks
b linenumber (, condition)
b file:linenumber (, condition)
b function name (, condition)
```

Breakpoints can be set with the b command. Specify a line number, or a function name, optionally preceded by the filename that contains it.

Any of the above can be followed by an expression (use comma to separate) to create a conditional breakpoint.

The **tbreak** command creates a one-time breakpoint that is deleted after it is hit the first time.

Profiling

- **Use the profile module from the command line**
- **Shows where program spends the most time**
- **Output can be tweaked via options**

Profiling is the technique of discovering the part of your code where your application spends the most time. It can help you find bottlenecks in your code that might be candidates for revision or refactoring.

To use the profiler, execute the following at the command line:

`python -m profile` *`scriptname.py`*

This will output a simple report to STDOUT.

Example

```
python -m profile count_with_dict.py
('eggs', 3)
('crumpets', 1)
('spam', 10)
         19 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
       14    0.000    0.000    0.000    0.000 :0(get)
        1    0.000    0.000    0.000    0.000 :0(items)
        1    0.000    0.000    0.000    0.000 :0(open)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000
count_with_dict.py:3(<module>)
        1    0.000    0.000    0.000    0.000 profile:0(<code
object <module> at 0xb74c36e0, file "count_with_dict.py", line
3>)
        0    0.000             0.000          profile:0(profiler)
```

The **pycallgraph** module (not in the standard library) will create a graphical representation on an application, indicating visually where the application is spending the most time.

Benchmarking

- **Use the** timeit **module**
- **Create a timer object with specified # of repetitions**

Use the **timeit** module to benchmark two or more code snippets. To time code, create a **Timer** object, which takes two strings of code. The first is the code to test; the second is setup code, that is only run once per timer .

Call the **timeit()** method with the number of times to call the test code, or call the **repeat()** method which repeats **timeit()** a specified number of times.

Example

timeit_ex.py

```
import timeit

setup_code = 'values = []'

test_code_one = '''
    for i in range(10000):
        values.append(i)
'''
test_code_two = '''
    i = 0
    while i < 10000:
        values.append(i)
        i += 1
'''

t1 = timeit.Timer(test_code_one, setup_code)
t2 = timeit.Timer(test_code_two, setup_code)

print("test one:")
print(t1.timeit(1000))
print()

print("test two:")
print(t2.timeit(1000))
print()
```

timeit_ex.py

```
test one:
1.1934709719998864

test two:
1.8586651270006769
```

Chapter 13 Exercises

Exercise 13-1

Pick several of your scripts (from class, or from real life) and run **pylint** on them.

Exercise 13-2 (testpresident.py)

Using the `unittest` module, write a test case to test the President class. Test that the first president's last name is Washington. Add any other tests you like.

Call the test case script **testpresident.py**

Exercise 13-3

Use any available debugger to step through one or more of the scripts you have written so far.

Chapter 14

XML and JSON

Chapter 14 Objectives

- **Understanding the layout of XML and JSON**
- **Using ElementTree to parse XML files**
- **Navigating the XML document tree**
- **Searching with XPath**
- **Creating and modifying XML with ElementTree**
- **Reading and writing JSON files**

About XML

- **Variant of SGML**
- **All data contained within tags**

An XML document consists of a single *element*, which contains sub-elements, which can have further sub-elements inside them. Elements are indicated by *tags* in the text. Tags are always inside angle brackets < >. Elements can either contain content, or they can be empty.

Tags can contain attributes, indicated by ***attribute="value".*** Tags can either appear in begin/end pairs, in which the end tag starts with a slash, or as a single tag, in which case the tag ends with a slash. Attributes must be surrounded by double quotes. All tag names and attribute names should be lower case.

Example
Solar.xml

```
<solarsystem>
    <star name="Sun"/>
    <innerplanets>
        <planet name="Mercury" type="inner"/>
        <planet name="Venus" type="inner"/>
        <planet name="Earth" type="inner">
            <moon>Moon</moon>
        </planet>
        <planet name="Mars" type="inner">
            <moon>Deimos</moon>
            <moon>Phobos</moon>
        </planet>
    </innerplanets>
    <asteroids>
        <asteroid >Ceres</asteroid>
        <asteroid >Pallas</asteroid>
        <asteroid >Juno</asteroid>
        <asteroid >Vesta</asteroid>
    </asteroids>
    <outerplanets>
        <planet name="Jupiter" type = "giant">
            <moon>Metis</moon>
            <moon>Adrastea</moon>
            <moon>Amalthea</moon>
            <moon>Thebe</moon>
        </planet
        ...
   </outerplanets>
</solarsystem>
```

Normal Approaches to XML

- **SAX**
  - **One scan through file**
  - **Good for large files**
  - **Uses callbacks on XML parsing events**
- **DOM**
  - **Builds a document tree**
  - **Python supports both through many libraries**

There are two approaches normally used in working with XML.

The first is called SAX, which stands for for Simple API for XML . It processes an XML file as a single stream, and so is appropriate for large XML files.

SAX processing consists of attaching callback functions to SAX events. These events are created when the XML reader encounters all the various components of an XML file – begin tags, end tags, data, and so forth.

The second approach is called the DOM, (Document Object Model), which parses an XML document into a tree that's fully resident in memory.

A top-level `Document` instance is the root of the tree, and has a single child which is the top-level `Element` instance; this `Element` has child nodes representing the content and any sub-elements, which may in turn have further children and so forth. Classes such as `Text`, `Comment`, `CDATASection`, `EntityReference`, provide access to XML structure and data. Nodes have methods for accessing the parent and child nodes, accessing element and attribute values, insert and delete nodes, and converting the tree back into XML.

The DOM is often useful for modifying XML documents, because you can create a DOM, modify it by adding new nodes and moving subtrees around, and then produce a new XML document as output.

While the DOM specification doesn't require that the entire tree be resident in memory at one time, many of the Python DOM implementations keep the whole tree in RAM, which can limit the size of the file being processed.

Which module to use?

- **Bewildering array of XML modules**
- **Some are SAX, some are DOM**
- **Use xml.etree.ElementTree!**

When you are ready to process Python with XML, you turn to the standard library, only to find a number of different modules with confusing names.

To cut to the chase, the xml.etree.ElementTree module is fast, provides both SAX and DOM style parsing, and unlike many of the other modules, has a Pythonic interface.

XML Modules in the Python 2.7 Standard Library

Module	Description
xml.parsers.expat	Fast XML parsing using Expat
xml.dom	The Document Object Model API
xml.dom.minidom	Lightweight DOM implementation
xml.dom.pulldom	Support for building partial DOM trees
xml.sax	Support for SAX2 parsers
xml.sax.handler	Base classes for SAX handlers
xml.sax.saxutils	SAX Utilities
xml.sax.xmlreader	Interface for XML parsers
xml.etree.ElementTree	The ElementTree XML API

ElementTree is great for simple XML tasks. If you need a faster or more full-featured xml library, use **lxml**. It is available from PyPI, and provides more features while maintaining an API compatible with ElementTree.

The examples in this course are designed to use **lxml** if available, and the builtin library.

Getting Started With XML

- **Import xml.etree.ElementTree or lxml.etree**
- **Create Elements as needed**
- **Create ElementTree for reading/writing**

xml.etree.ElementTree is part of the Python standard library. **lxml** is available from PyPI, or as part of the Anaconda bundle.

To save typing , it is typical to alias one of he above modules as **ET.**

Example

```
HAS_LXML = False
try:
    import lxml.etree as ET
    HAS_LXML = True
except ImportError:
    import xml.etree.ElementTree as ET
```

How ElementTree Works

- **One ElementTree to contain the document**
- **Document is a tree of Elements**
- **Each Element has**
  - **Tag name**
  - **Attributes (implemented as a dictionary)**
  - **Text**
  - **Tail**
  - **Child elements (implemented as a list) (if any)**

In **ElementTree,** an XML document consists of a nested tree of **Element** objects. Each **Element** corresponds to an XML tag. An **ElementTree** object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use ElementTree.parse(); this creates the **ElementTree** wrapper and the tree of Elements. You can then navigate to, or search for, **Elements** within the tree. You can also insert and delete new elements.

If you are creating a new document from scratch, create a top-level (AKA "root") element, then create child elements as needed.

When creating a new Element, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an ElementTree with the root element.

The **Element** object is a hybrid of list and dictionary. You access child elements by treating it as a list. You access attributes by treating it as a dictionary. (But you can't use subscripts for the attributes – you must use the **get()** method).

```
element = root.find('sometag')
for subelement in element:
    print subelement.tag
print element.get('someattribute')
```

The Element object also has several useful properties: **tag** is the element's tag; **text** is the text contained inside the element; **tail** is the text following the element, before the next element.

Only the *tag* property of Element is required

Element properties and methods	
Property	**Description**
append*(element)*	Add a subelement *element* to end of subelements
attrib	Dictionary of element's attributes
clear()	Remove all subelements
find(path)	Find first subelement matching *path*
findall(*path*)	Find all subelements matching *path*
findtext(*path*)	Shortcut for find(*path*).text
get(*attr*)	Get an attribute; Shortcut for attrib.get()
getiterator()	Returns an iterator over all descendants
getiterator(*path*)	Returns an iterator over all descendants matching *path*
insert(pos,element)	Insert subelement *element* at position *pos*
items()	Get all attribute values; Shortcut for attrib.items()
keys()	Get all attribute names; Shortcut for attrib.keys()
remove(*element*)	Remove subelement *element*
set(*attrib*,*value*)	Set an attribute value; shortcut for attr*[attrib] = value*
tag	The element's tag
tail	Text following the element
text	Text contained within the element

ElementTree properties and methods	
Property	**Description**
find(*path*)	Finds the first toplevel element with given tag; shortcut for getroot().find(path).
findall(*path*)	Finds all toplevel elements with the given tag; shortcut for getroot().findall(path).
findtext(*path*)	Finds element text for first toplevel element with given tag; shortcut for getroot().findtext(path).
getiterator*(path)*	Returns an iterator over all descendants of root node matching *path*. (All nodes if *path* not specified)
getroot()	Return the root node of the document
parse(*filename*) parse(*fileobj*)	Parse an XML source (filename or file-like object)
write(*filename*,*encoding*)	Writes XML document to *filename*, using *encoding* (Default **us-ascii**).

Creating a New XML Document

- **Create root element**
- **Add descendants**
- **Use keyword arguments for attributes**
- **Add text after element created**
- **Create ElementTree for import/export**

To create a new XML document, first create the root (top-level) element. This will be a container for all other elements in the tree. If your XML document contains books, for instance, the root document might use the "books" tag. It would contain one or more "book" elements, each of which might contain author, title, and ISBN elements.

Once all the elements are in place, you can create an ElementTree object to contain the elements and allow you to write out the XML text.

Example

create_xml_doc

```
from xml.etree import ElementTree as ET

root = ET.Element('movies')

movie1 = ET.Element('movie', director='Spielberg, Stephen')
root.append(movie1)
movie1.text = 'Jaws'

movie2 = ET.Element('movie',director='Hitchcock, Alfred')
movie2.text = 'Vertigo'
actor1 = ET.Element('actor')
actor1.text = 'James Stewart'
movie2.append(actor1)
actor2 = ET.Element('actor')
actor2.text = 'Kim Novak'
movie2.append(actor2)
root.append(movie2)

movie3 = ET.Element('movie',director='Welles, Orson')
movie3.text = 'Citizen Kane'
root.append(movie3)

doc = ET.ElementTree(root)

doc.write('movies.xml')
```

movies.xml

```
<movies><movie director="Spielberg, Stephen">Jaws</movie><movie
director="Hitchcock, Alfred">Vertigo<actor>James
Stewart</actor><actor>Kim Novak</actor></movie><movie director="Welles,
Orson">Citizen Kane</movie></movies>
```

Parsing An XML Document

- **Use ElementTree.parse()**
- **returns an ElementTree object**
- **Use get... or find... methods to select element**

Use the **parse()** method to parse an existing XML document. It returns an **ElementTree** object, from which you can find the root, or any other element within the document.

To get the root element, use the **getroot()** method.

The **ElementTree** object also supports the **get...** and **find...** methods of the **Element** object.

Navigating the XML Document

- **Start at root or use find... methods**
- **Each Element is also an iterable of its subelements**

To find the first child element with a given tag, use find('tag'). The findtext('tag') method is the same, but returns the text within the tag.

To loop through all elements with a given tag, use the findall('tag') method.

Use care when checking for child elements. Say

`if node is None:`

to see whether a node was found, but

`if len(node) > 0:`

to see whether a node has children.

A node with no children tests as false because it is an empty list, but it is not None.

Example

planet_nav.pl

```
import sys
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

root = doc.getroot()

inner = root.find('solarsystem').find('innerplanets')
outer = root.find('solarsystem').find('outerplanets')

print 'Inner:'
for planet in inner:  # loop through children
    print '\t',planet.get("name")

print 'Outer:'
for planet in outer:  # loop through children
    print '\t',planet.get("name")
```

planet_nav.pl

```
Inner:
          Mercury
          Venus
          Earth
          Mars
Outer:
          Jupiter
          Saturn
          Uranus
          Neptune
```

Example

read_movies.py

```
import xml.etree.ElementTree as ET

movies_doc = ET.parse('movies.xml')

movies_root = movies_doc.getroot()

for movie_node in movies_root:
    print movie_node.text
    for actor in movie_node:
        print '\t',actor.text
```

```
read_movies.py
Jaws
Vertigo
          James Stewart
          Kim Novak
Citizen Kane
```

Using XPath

- **Use simple XPath patterns**
- **Works with find... methods**

When a simple tag is specified, the **find...** methods only search for subelements of the current element. For more flexible searching, the **find...** methods work with simplified XPath patterns. To find grandchildren, for instance, use the XPath pattern:

```
movies/movie/actor
presidents/president/name/last
```

The patterns available include *, . and // as well.

Asterisk (*) matches all tags at the specified level: **/*/planets**

Period (.) matches the current node: **.//planets**

Double slash (//) searches all levels beneath current element: **//lname**

Example

planets_xpath1.py

```
import sys
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

inner_nodes = doc.findall('solarsystem/innerplanets/planet')

outer_nodes = doc.findall('solarsystem/outerplanets/planet')

print 'Inner:'
for planet in inner_nodes:  # + outer_nodes:
    print '\t',planet.get("name")

print 'Outer:'
for planet in outer_nodes:  # + outer_nodes:
    print '\t',planet.get("name")
```

```
planets_xpath1.py
Inner:
          Mercury
          Venus
          Earth
          Mars
Outer:
          Jupiter
          Saturn
          Uranus
          Neptune
```

planets_xpath2.py

```
import sys
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

planets = doc.findall('solarsystem/*/planet')

for planet in planets:
    print planet.get("name")
```

```
planets_xpath2.py
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto
```

Advanced Xpath

- **More flexible searching**
- **Specify attributes and values**
- **Wild cards**
- **Starting with ElementTree v1.3**

Beginning with version 1.3, the **find...** methods of an Element support some new features for searching. These are a more complete implementation of XPath.

See the chart on the next page for details.

ElementTree XPath Summary	
Syntax	**Meaning**
tag	Selects all child elements with the given tag. For example, “spam” selects all child elements named “spam”, “spam/egg” selects all grandchildren named “egg” in all child elements named “spam”. You can use universal names (“{url}local”) as tags.
*	Selects all child elements. For example, “*/egg” selects all grandchildren named “egg”.
.	Select the current node. This is mostly useful at the beginning of a path, to indicate that it’s a relative path.
.//	Selects all subelements, on all levels beneath the current element (search the entire subtree). For example, “.//egg” selects all “egg” elements in the entire tree.
..	Selects the parent element.
[@attrib]	Selects all elements that have the given attribute. For example, “.//a[@href]” selects all “a” elements in the tree that has a “href” attribute.
[@attrib=’value’]	Selects all elements for which the given attribute has the given value. For example, “.//div[@class=’sidebar’]” selects all “div” elements in the tree that has the class “sidebar”. In the current release, the value cannot contain quotes.
[tag]	Selects all elements that has a child element named tag. In the current version, only a single tag can be used (i.e. only immediate children are supported).
[position]	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression “last()” (for the last position), or a position relative to last() (e.g. “last()-1” for the second to last position). This predicate must be preceeded by a tag name.
Note: Tags in gray only available in ElementTree version 1.3 and later (Python 2.7+)	

About JSON

- **Lightweight, human-friendly format for data**
- **Contains dictionaries and lists**
- **Stands for Javascript Object Notation**
- **Looks like Python!**
- **Basic types: Number, String, Boolean, Array, Object**
- **White space is ignored**

JSON is a lightweight and human-friendly format for sharing or storing data. It was developed and popularized by Douglas Crockford starting in 2001.

A JSON file contains *objects* and *arrays*, which correspond exactly to Python dictionaries and lists. In fact, JSON may be directly parsed by the Python interpreter.

White space is ignored, so JSON may be formatted for readability.

Data types are Number, String, and Boolean. Strings are enclosed in double quotes (only); numbers look like integers or floats; booleans are represented by **true** or **false;** null is represented by **null**.

Example

json_equiv.py

```
x = {
     "firstName": "John",
     "lastName" : "Smith",
     "age"      : 25,
     "smoker"   : false,
     "address"  :
     {
         "streetAddress": "21 2nd Street",
         "city"         : "New York",
         "state"        : "NY",
         "postalCode"   : "10021"
     },
     "phoneNumber":
     [
         {
           "type"  : "home",
           "number": "212 555-1234"
         },
         {
           "type"  : "fax",
           "number": "646 555-4567"
         }
     ]
 }
```

Reading JSON

- json **module part of standard library**
- **API similar to** pickle **and** marshall **modules**
- **json.loads() parse from string**
- **json.load() parse from file-like object**
- **Both methods return Python dict**

To read a JSON file, import the **json** module. Use **json.loads()** to parse a string containing valid JSON. Use **json.load()** to read JSON from a file-like object[1].

Both methods return a Python dictionary containing all the data from the JSON file.

Example

json_read_ex.py

```
import json

with open('../DATA/solar.json') as SOLAR:
    solar = json.load(SOLAR)

for p in solar['innerplanets']:
    print(p['name'])

for p in solar['outerplanets']:
    print(p['name'])

for p in solar['dwarfplanets']:
    print(p['name'])
```

```
json_read_ex.py
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto
```

1Any object that acts like a file – i.e., implements read(), readline(), etc.

Writing JSON

- **Use json.dumps() or json.dump()**

To output JSON to a string, use json.dumps(). To output JSON to a file, pass a file-like object to json.dump().

Example

json_write_ex.py

```
import json

george = {
    'num':1,
    'lname':'Washington',
    'fname':'George',
    'dstart':[1789,4,30],
    'dend':[1797,3,4],
    'birthplace':'Westmoreland County',
    'birthstate':'Virginia',
    'dbirth':[1732,2,22],
    'ddeath':[1799, 12,14],
    'assassinated': False,
    'party':'no party',
}

js = json.dumps(george)
print(js)

with open('george.json','w') as JS:
    json.dump(george,JS)
```

json_write_ex.py

```
{"dstart": [1789, 4, 30], "ddeath": [1799, 12, 14], "num": 1,
"birthstate": "Virginia", "dbirth": [1732, 2, 22], "birthplace":
"Westmoreland County", "lname": "Washington", "fname": "George",
"party": "no party", "assassinated": false, "dend": [1797, 3, 4]}
```

The JSON produced by **json** (with default settings) is a subset of YAML, so it may be used as a serializer for that as well.

Chapter 14 Exercises

Exercise 14-1

Using **ElementTree,** create a new XML file containing all the words that start with 'x' from words.txt. The root tag should be named 'words', and each word should be contained in a 'word' tag. The finished file should look like this:

```
<words>
   <word>xanthan</word>
   <word>xanthans</word>
   and so forth
</words>
```

Name this script **xwords.py**.

Exercise 14-2

Use **ElementTree** to parse **presidents.xml**. Loop through and print out each president's first and last names and their state of birth. Name this script **xpres.py**.

Exercise 14-3

Repeat exercise 14-2, but parse **presidents.json** using the **json** module. Name the script **jpres.py**.

Chapter 15

iPython

Chapter 15 Objectives

- **Learn the basics of iPython**
- **Use and apply *magics***
- **Understand collaboration via iPython notebooks**
- **Create and display graphics in iPython**

About iPython

- **Enhanced interpreter for exploratory computing**
- **Efficient for data analysis (what-if? Scenarios)**
- **Great for plotting-tweaking-plotting**
- **Good for simple code development**
- **Not so good for enterprise development**

iPython is an enhanced interpreter for Python. It provides a large number of "creature comforts" for the user, such as name completion and improved help features.

It's great for ad-hoc queries, what-if scenarios, when you are not developing a full application.

Features of iPython

- **Name completion (variables, modules, methods, etc.)**
- **Autoindent**
- **Inline plots and other figures**
- **Dynamic introspection (dir() on steroids)**
- **Search namespaces with wildcards**
- **Auto-parentheses ('sin 3' becomes 'sin(3)'**
- **Auto-quoting (',foo a b' becomes 'foo("a","b")'**
- **Enhanced help system**
- **Commands are numbered (and persistent) for recall**
- **'Magic' commands for controlling iPython itself**
- **Aliasing system for interpreter commands**
- **Easy access to shell commands**
- **Background execution in separate thread**
- **Simplified (and lightweight) persistence**
- **Session logging (can be saved as scripts)**
- **Detailed tracebacks when errors occur**
- **Session restoring (playback log to specific state)**
- **Flexible configuration system**
- **Easy access to Python debugger**
- **Simple profiling**
- **Interactive parallel computing (if supported by hardware)**

The many faces of iPython

- **Console**
- **Colorized console**
- **QTConsole**
- **Web notebook**

iPython can be run in several different modes. The most basic mode is *console*, which runs from the command prompt, without syntax colorizing.

If your terminal supports it (and most do), **iPython** can run as a colorized console, using different colors for variables, functions, strings, comments, and so forth.

Both the default console and the colorized console display plots in a separate popup window.

If QT is installed, **iPython** can run a GUI console, which looks and acts like a regular text console, but allows plots to be generated inline, and has enhanced context-sensitive help.

The most flexible and powerful way to run iPython is in *notebook* mode. This mode starts a dedicated web server and begins a session using your default web browser. Other users can load notebooks from the notebook server, similarly to MatLab.

iPython also supports parallel computing, which will not be covered here.

Starting iPython

- **Type ipython at the command line**
- **Add --pylab to import numpy and matplotlib as plt**
- **Commands are *console*, *qtconsole*, and *notebook***
- **Keyword *inline* may be added after options**
- **Huge number of options**

To get started with iPython in console mode, just type **ipython** at the command line, or double-click the icon in Windows explorer.

In general, it works like the normal interpreter, but with many more features.

Add the **--color-info** option to provide colorized syntax. This may not work in some text consoles. It should work fine with **qtconsole**. It is on by default in notebook mode.

Add the **--pylab** option to import **numpy** as **np**, **scipy** as **sp**, and **matplotlib.pyplot** as **plt**.

There are several keywords that specify the mode in which **iPython** starts up.

There is a huge number of options. To see them all, invoke **iPython** with the --help-all option:

```
ipython –help-all
```

iPython Startup	

Console ("normal")	`ipython --pylab`
	or
	`ipython console --pylab`

Color console	`ipython --pylab –color-info`
QT console	`ipython qtconsole --pylab`
Notebook	`ipython notebook --pylab`

For QT console and notebook mode, the **inline** command can be added, to display plots inline:

```
ipython –pylab notebook inline
```

Without **inline**, plots are displayed in a popup window.

Getting Help

- **?** **basic help**
- ***%*quickref** **quick reference**
- **help** **standard Python help**
- ***thing*?** **help on thing**

iPython provides help in several ways.

Typing **?** at the prompt will display an introduction to iPython and a feature overview.

For a quick reference card, type ***%*quickref**.

To start Python's normal help system, type **help**.

For help on any Python object, type ***object*?**. This is similar to saying help("object") in the default interpreter, but is "smarter".

For more help on an object, add a second question mark:

Tab Completion

- **Press tab to complete**
- **keywords**
- **modules**
- **methods and attributes**
- **parameters to functions**
- **file and directory names**

Pressing the <TAB> key will invoke autocomplete. If there is only one possible comlpletion, it will be expanded. If there is more than one completion that will match, iPython will display a list of possible completions.

Autocomplete candidates include keywords, functions, classes, methods, and object attributes, as well as paths from your file system.

Cells

- **Basic units of execution**
- **May be executed individually**
- **Numbered by iPython**
- **Three kinds of cells**
  - **– code**
  - **– documentation (notebooks only)**
  - **– plain text (notebooks only)**

The basic unit of code in iPython is a *cell*. A cell can contain one or more lines of code. A cell can be executed as a unit, separately from the rest of the code.

To create a cell in console mode, type a cell magic and hit enter.

In notebook mode, cells are placed in separate blocks.

Magic Commands

- **Start with % (line magic) or %% (cell magic)**
- **Simplify common tasks**
- **Use %lsmagic to list all magic commands**

One of the enhancements in **iPython** is the set of "magic" commands. These are meta-commands that help you manipulate the iPython environment.

Normal magics apply to a single line. Cell magics apply to a cell (a group of lines).

For instance, **%history** will list previous commands.

See the **iPython Quick Reference** (or type **%lsmagic**) for a list of all the magics

Benchmarking

- **Use %timeit**

iPython has a handy magic for benchmarking.

```
In [1]: color_values = { 'red':66, 'green':85, 'blue':77 }

In [2]: %timeit red_value = color_values['red']
10000000 loops, best of 3: 54.5 ns per loop

In [3]: %timeit red_value = color_values.get('red')
10000000 loops, best of 3: 115 ns per loop
```

will benchmark whatever code comes after it on the same line.
%%timeit will benchmark multiple lines.

External commands

- **Precede command with !**

Any shell command can be run by starting it with a !.

Windows

```
In [3]: !dir DATA\*.csv
 Volume in drive Z is Shared Folders
 Volume Serial Number is 0000-0064

 Directory of Z:\Desktop\py4sci\DATA

02/20/2014  01:53 PM             5,511 airport_boardings.csv
02/20/2014  01:53 PM             2,182 energy_use_quad.csv
02/20/2014  01:53 PM             4,993 parasite_data.csv
               3 File(s)         12,686 bytes
               0 Dir(s)  352,625,324,032 bytes free

In [4]:
```

Non-Windows (Linux, OS X, etc)

```
In [2]: !ls -l DATA/*.csv
-rwxr-xr-x  1 jstrick  staff  5511 Jan 27 19:44 DATA/airport_boardings.csv
-rwxr-xr-x  1 jstrick  staff  2182 Jan 27 19:44 DATA/energy_use_quad.csv
-rwxr-xr-x  1 jstrick  staff  4642 Jan 27 19:44 DATA/parasite_data.csv

In [3]:
```

Enhanced help

- **Use object?**
- **Output of dir() is cleaner**
- **Multidimensional data structures are prettyprinted**

iPython makes it easier to get at metadata. Type an object's name followed by a question mark will provide a description of the object plus any available help. It is similar to, but much better than, the help() function in the standard interpreter.

Dictionaries and multidimensional data structures are prettyprinted – items are printed on separate lines.

Notebooks

- **Puts the interpreter in a web browser**
- **Code is grouped into "cells"**
- **Cells can be edited, repeated, etc.**

One of the most powerful extensions to iPython is the *notebook*. The notebook is a journal-like python interpreter that lives in a browser window. Code is grouped into *cells*, which can contain multiple statements. Cells can be edited, repeated, rearranged, and otherwise manipulated.

Plots will be shown inline.

A notebook (i.e, a set of cells, can be saved, and reopened). Notebooks can be shared among members of a team via the notebook server which is built into **ipython**.

To start iPython in notebook/pylab mode, issue the following command:

```
ipython notebook -pylab
```

*At this point please start iPython in notebook mode, open **ipython_demo.ipynb** and follow along with the instructor (start in the root of the student files):*

```
cd NOTEBOOKS
ipython notebook -pylab
```

Inline Plots

- **Plots can be inline or popup**
- **Inline stay with the notebook**
- **Popups have a frame with tools**
- **Add inline keyword when starting ipython**

When you generate a plot, you have a choice of displaying the plot *inline* (in a notebook cell) or in a popup window.

Inline plots are part of the notebook and will be visible as soon as someone else loads the notebook. Popup plots have a frame with some tools for manipulating

Sharing Notebooks

- **Use "public" IP address (not 127.0.0.1)**
- **Start notebook server (local browser starts)**
- **Point remote browser to server**
- **Use port 8888 by default**

By default, the notebook server starts on IP address 127.0.0.1, which is only accessible on the local computer. To share a notebook with a user on a different computer, start the notebook server on the LAN IP address with the –ip option:

```
ipython notebook --pylab --ip=192.168.18.1
```

Then you can use a browser on any machine which can access that IP address, using the port 8888:

http:///192.168.18.1:8888

Anyone with a notebook open can save it with their current changes. For others to see the changes, they need to reopen the notebook.

Chapter 16

NumPy

Chapter 16 Objectives

- **See the "big picture" of numpy**
- **Create and manipulate arrays**
- **Learn different ways to initialize arrays**
- **Understand the numpy data types available**
- **Work with shapes, dimensions, and ranks**
- **Broadcast changes across multiple array dimensions**
- **Extract multidimensional slices**
- **Perform matrix operations**

Python's scientific stack

- **NumPy, SciPy, MatPlotLib (and many others)**
- **Python extensions, written in C/Fortran**
- **Support for math, numerical, and scientific operations**

NumPy is part of what is sometimes called Python's "scientific stack". Along with SciPy, MatPlotLib, and other libraries, it provides a broad range of support for scientific and engineering tasks.

There is no one integrated tool for these libraries. You can either create scripts with your choice of IDE, or you can use iPython to manipulate data interactively and *ad hoc*.

NumPy overview

- **Install numpy module from** numpy.scipy.org
- **Basic object is the array**
- **Up to 100x faster than normal Python math operations**
- **Functional-based (fewer loops)**
- **Dozens of utility functions**

The basic object that **NumPy** provides is the *array*. Arrays can have as many dimensions as needed. Working with NumPy arrays can be 100 times faster than working with normal Python lists.

Operations are applied to arrays in a functional manner – instead of the programmer explicitly looping through elements of the array, the programmer specifies an expression or function to be applied, and the array object does all of the iteration internally.

There are many utility functions for accessing arrays, for creating arrays with specified values, and for performing standard numerical operations on arrays.

To get started, import the **numpy** module. It is conventional to import numpy as **np**. The examples in this chapter will follow that convention.

Creating Arrays

- **Create with**
- **array() function initialized with nested sequences**
- **Other utilities (arange(), zeros(), ones(), empty()**
- **r_, c_ objects**
- **All elements are same type (default float)**
- **Useful properties: ndim, shape, size, dtype**
- **Can have any number of axes (dimensions)**
- **Each axis has a length**
- **Rank is the # axes**

An array is the most basic object in numpy. It is a table of numbers, indexed by positive integers. All of the elements of an array are of the same type.

An array can have any number of dimensions; these are referred to as axes. The number of axes is called the rank.

Arrays are rectangular, not ragged.

One way to create an array is with the **array()** function, which can be initialized from a sequence of sequences.

The **zeros()** function expects a list of axis lengths, and creates the corresponding array, with all values set to zero. The **ones()** function is the same, but initializes with ones.

The **empty()** function creates an array initialized with random floats.

When you print an array, NumPy displays it with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

Example

np_create_arrays.py

```
import numpy as np

# from nested sequences
a_nest = np.array([[1,2,3],[4,5,6],[7,8,9]])
print a_nest
print

# with zeros
a_zeros = np.zeros([3,5])
print a_zeros
print

# with ones
a_ones = np.ones([2,3,4,5])
print a_ones
print

# with uninitialized values
a_empty = np.empty([3,8])
print a_empty
```

np_create_arrays.py

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]

[[[[ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]]

  [[ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]]

  [[ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]]]


 [[[ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]]

  [[ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]]

  [[ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]
   [ 1.  1.  1.  1.  1.]]]]

[[  1.49166815e-154   2.00000012e+000   3.64219380e-258   2.80027053e-309
    2.12339378e-314   2.12347894e-314   1.45857511e-308   7.05159694e-278]
 [  1.49166815e-154   1.49166815e-154   1.37929726e-312   8.34402697e-309
    1.49166815e-154   2.00000012e+000   1.50562780e-316   2.11382186e-307]
 [  1.49166815e-154  -2.31584192e+077   2.12439184e-314   5.56270602e-309
    2.12339378e-314   2.13505857e-314   4.22765225e-307   1.66880753e-308]]
```

Creating ranges

- **Similar to builtin xrange()**
- **Returns a one-dimensional numpy array**
- **Can use floating point values**
- **Can be reshaped**

The arange() function takes a size, and returns a one-dimensional **numpy** array. This array can then be reshaped as needed. The start, stop, and step parameters are similar to those of xrange().

The linspace() function creates a specified number of equally-spaced values.

The resulting arrays can be reshaped into multidimensional arrays.

Example

np_create_ranges.py

```
import numpy as np

r1 = np.arange(50)
print r1
print "size is", r1.size
print

r2 = np.arange(5,101,5)
print r2
print "size is", r2.size
print

r3 = np.arange(1.0,5.0,.33)
print r3
print "size is", r3.size
print

r4 = np.linspace(1.0,5.0,16)
print r4
print "size is", r4.size
print
```

np_create_ranges.py

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
size is 50

[  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90
  95 100]
size is 20

[ 1.    1.33  1.66  1.99  2.32  2.65  2.98  3.31  3.64  3.97  4.3   4.63
  4.96]
size is 13

[ 1.          1.26666667  1.53333333  1.8         2.06666667  2.33333333
  2.6         2.86666667  3.13333333  3.4         3.66666667  3.93333333
  4.2         4.46666667  4.73333333  5.        ]
size is 16
```

Working with arrays

- **Use normal math operators (+, -, /, and *)**
- **Use NumPy's builtin functions**
- **By default, apply to every element**
- **Can apply to single axis**
- **Operations on between two arrays applies operator to pairs of element**

The array object is smart about applying functions and operators. A function applied to an array is applied to every element of the array. An operator applied to two arrays is applied to corresponding elements of the two arrays.

Some builtin array functions

all, alltrue, any, apply along axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, conjugate, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, inv, lexsort, max, maximum, mean,median, min, minimum, nonzero, outer, prod, re, round, sometrue, sort, std, sum, trace, transpose, var, vdot, vectorize, where

Example

np_basic_array_ops.py

```
import numpy as np

a = np.array(
    [
        [ 5, 10, 15 ],
        [ 2,  4, 6  ],
        [ 3,  6, 9, ],
    ]
)

b = np.array (
    [
        [ 10, 85, 92 ],
        [ 77, 16, 14 ],
        [ 19, 52, 23 ],
    ]
)
print "a"
print a
print

print "b"
print b
print

print "a * 10"
print a * 10
print

print "a + b"
print a + b
print

print "b + 3"
print b + 3
print

s1 = a.sum()
s2 = b.sum()
print "sum of a is {0}; sum of b is {1}".format(s1,s2)
```

np_basic_array_ops.py

```
a
[[ 5 10 15]
 [ 2  4  6]
 [ 3  6  9]]

b
[[10 85 92]
 [77 16 14]
 [19 52 23]]

a * 10
[[ 50 100 150]
 [ 20  40  60]
 [ 30  60  90]]

a + b
[[ 15  95 107]
 [ 79  20  20]
 [ 22  58  32]]

b + 3
[[13 88 95]
 [80 19 17]
 [22 55 26]]

sum of a is 60; sum of b is 388
```

Shapes

- **Number of elements on each axis**
- ***array*.shape has shape tuple**
- **Assign to array.shape to change**
- ***array*.ravel() to flatten to one dimension**
- ***array*.transpose() to flip the shape**

Every array has a *shape*, which is the number of elements on each axis. For instance, an array might have the shape (3,5), which means that there are 3 rows and 5 columns.

The shape is stored as a tuple, in the **shape** attribute of an array. To change the shape of an array, assign to the **shape** attribute.

The **ravel()** method will flatten any array into a single dimension. The **transpose()** method will flip shape (x,y) to shape (y,x). It is equivalent to *array*.shape = reversed(*array*.shape).

Example

np_shapes.py

```
import numpy as np

a1 = np.arange(15)
print "a1 shape",a1.shape
print

print a1
print

a1.shape = 3,5
print a1
print

a1.shape = 5,3
print a1
print

print a1.flatten()
print

print a1.transpose()
print "------------------"

a2 = np.arange(40)
a2.shape = 2,5,4

print a2
print

print a2.transpose()
print a2.transpose().shape
print

a2.shape = 2,2,10
print a2
print
```

np_shapes.py

```
a1 shape (15,)

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  3  6  9 12]
 [ 1  4  7 10 13]
 [ 2  5  8 11 14]]
------------------
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]
  [12 13 14 15]
  [16 17 18 19]]

 [[20 21 22 23]
  [24 25 26 27]
  [28 29 30 31]
  [32 33 34 35]
  [36 37 38 39]]]

[[[ 0 20]
  [ 4 24]
  [ 8 28]
  [12 32]
  [16 36]]

 [[ 1 21]
  [ 5 25]
  [ 9 29]
  [13 33]
  [17 37]]

 [[ 2 22]
  [ 6 26]
  [10 30]
  [14 34]
  [18 38]]

 [[ 3 23]
  [ 7 27]
  [11 31]
  [15 35]
  [19 39]]]
(4, 5, 2)

[[[ 0  1  2  3  4  5  6  7  8  9]
  [10 11 12 13 14 15 16 17 18 19]]

 [[20 21 22 23 24 25 26 27 28 29]
  [30 31 32 33 34 35 36 37 38 39]]]
```

Slicing and indexing

- **Simple indexing similar to lists**
- **start, stop, step**
- **start is INclusive, stop is Exclusive**
- **: used for range for one axis**
- **... means "as many : as needed"**

NumPy arrays can be indexed and sliced like regular Python lists, but with some convenient extensions. Instead of [x][y], numpy arrays can be indexed with [x,y]. Within an axis, ranges can be specified with slice notation (*start:stop:step*) as usual.

For arrays with more than 2 dimensions, '...' can be used to mean "and all the other dimensions".

Example

np_indexing.py

```
import numpy as np

a = np.array(
[[70, 31, 21, 76, 19, 5, 54, 66],
 [23, 29, 71, 12, 27, 74, 65, 73],
 [11, 84, 7, 10, 31, 50, 11, 98],
 [25, 13, 43, 1, 31, 52, 41, 90],
 [75, 37, 11, 62, 35, 76, 38, 4]]
)

print a
print

print 'a[0] =>', a[0]              # first row
print 'a[0][0] =>', a[0][0]        # first element of first row
print 'a[0,0] =>', a[0,0]          # same, but numpy style
print 'a[0,:3] =>', a[0,:3]        # first 3 elements of first row
print 'a[0,::2] =>', a[0,::2]      # every second element of first row
print
print 'a[::2] =>', a[::2]          # every second row
print
print 'a[::2,::3] =>', a[::2,::3] # every 3rd element of every 2nd row
```

np_indexing.py

```
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

a[0] => [70 31 21 76 19  5 54 66]
a[0][0] => 70
a[0,0] => 70
a[0,:3] => [70 31 21]
a[0,::2] => [70 21 19 54]

a[::2] => [[70 31 21 76 19  5 54 66]
 [11 84  7 10 31 50 11 98]
 [75 37 11 62 35 76 38  4]]

a[::2,::3] => [[70 76 54]
 [11 10 11]
 [75 62 38]]
```

Indexing with Booleans

- **Apply relational expression to array**
- **Result is array of Booleans**
- **Booleans can be used to index original array**

If a relational expression (>, <, >=, <=) is applied to an array, the result is a new array containing booleans reflecting whether the expression was true for each element. That is, for each element of the original array, the resulting array is set to True if the expression is true for that element, and False otherwise.

The resulting Boolean array can then be used as an index, to modify just the elements for which the expression was true.

Example

np_bool_indexing.py

```
import numpy as np

a = np.array(
[[70, 31, 21, 76, 19, 5, 54, 66],
 [23, 29, 71, 12, 27, 74, 65, 73],
 [11, 84, 7, 10, 31, 50, 11, 98],
 [25, 13, 43, 1, 31, 52, 41, 90],
 [75, 37, 11, 62, 35, 76, 38, 4]]
)

print 'a =>', a
print

i = a > 50
print 'i (a > 50) =>', i
print

print 'a[i] =>', a[i]
print
print 'a[i].min(), a[i].max() =>', a[i].min(), a[i].max()
print
a2 = np.copy(a)
a2[i] = 0
print 'a2 =>', a2
print
```

np_bool_indexing.py

```
a => [[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

i (a > 50) => [[ True False False  True False False  True  True]
 [False False  True False False  True  True  True]
 [False  True False False False False False  True]
 [False False False False False  True False  True]
 [ True False False  True False  True False False]]

a[i] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[i].min(), a[i].max() => 52 98

a2 => [[ 0 31 21  0 19  5  0  0]
 [23 29  0 12 27  0  0  0]
 [11  0  7 10 31 50 11  0]
 [25 13 43  1 31  0 41  0]
 [ 0 37 11  0 35  0 38  4]]
```

Stacking

- **Combining 2 arrays vertically or horizontally**
- **use vstack() or hstack()**
- **Arrays must have compatible shapes**

You can combine 2 or more arrays vertically or horizontally with the vstack() or hstack() functions. These functions are also handy for adding rows or columns with the results of operations.

Example

np_stacking.py

```
import numpy as np

a = np.array(
[[70, 31, 21, 76, 19, 5, 54, 66],
 [23, 29, 71, 12, 27, 74, 65, 73]]
)

b = np.array(
[[11, 84, 7, 10, 31, 50, 11, 98],
 [25, 13, 43, 1, 31, 52, 41, 90]]
)

print 'a =>\n', a
print
print 'b =>\n', b
print
print 'vstack((a,b)) =>\n', np.vstack((a,b))
print

print 'vstack((a,a[0] + a[1])) =>\n', np.vstack((a,a[0] + a[1]))
print

print 'hstack((a,b)) =>\n', np.hstack((a,b))
```

np_stacking.py

```
a =>
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]]

b =>
[[11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,b)) =>
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,a[0] + a[1])) =>
[[ 70  31  21  76  19   5  54  66]
 [ 23  29  71  12  27  74  65  73]
 [ 93  60  92  88  46  79 119 139]]

hstack((a,b)) =>
[[70 31 21 76 19  5 54 66 11 84  7 10 31 50 11 98]
 [23 29 71 12 27 74 65 73 25 13 43  1 31 52 41 90]]
```

Iterating

- **Similar to normal Python**
- **Iterates through first dimension**
- **Use *array*.flat to iterate through all elements**

Iterating through a NumPy array is similar to iterating through any Python list; iteration is across the first dimension. Slicing and indexing can be used.

To iterate across every element, use the **flat** property of the array.

Example

np_iterating.py

```
import numpy as np

a = np.array(
[[70, 31, 21, 76, 19, 5, 54, 66],
 [23, 29, 71, 12, 27, 74, 65, 73]]
)

b = np.array(
[[11, 84, 7, 10, 31, 50, 11, 98],
 [25, 13, 43, 1, 31, 52, 41, 90]]
)
print 'a =>\n', a
print

print 'b =>\n', b
print

print "for row in a: =>"
for row in a:
    print "row:", row
print

print "for elem in a.flat: =>"
for elem in a.flat:
    print "element:", elem
```

np_iterating.py

```
a =>
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]]

b =>
[[11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

for row in a: =>
row: [70 31 21 76 19  5 54 66]
row: [23 29 71 12 27 74 65 73]

for elem in a.flat: =>
element: 70
element: 31
element: 21
element: 76
element: 19
element: 5
element: 54
element: 66
element: 23
element: 29
element: 71
element: 12
element: 27
element: 74
element: 65
element: 73
```

Array creation shortcuts

- **Use "magic" arrays r_, c_**
- **Either creates a new numpy array**
- **Index values determine resulting array**
- **List of arrays creates a "stacked" array**
- **List of values creates a 1D array**
- **Slice notation creates a range of values**
- **A complex step creates equally-spaced value**

Numpy provides several shortcuts for working with arrays.

The **r_** object can be used to magically build arrays via its index expression. It acts like a magic array, and "returns" (evaluates to) a normal numpy ndarray object.

There are two main ways to use r_():

If the index expression contains a list of arrays, then the arrays are "stacked" along the first axis.

If the index contains slice notation, then it creates a one-dimensional array, similar to numpy.arange(). It uses *start*, *stop*, and *step* values. However, if *step* is an imaginary number (a literal than ends with 'j'), then it specifies the number of points wanted, more like **numpy.linspace()**.

There can be more than one slice, as well as individual values, and ranges. They will all be concatenated into one array.

The first element in the index can be a either a string that modifies how the array is created, or a string that makes the result a **matrix** instead of an **ndarray**.

If the first element is a string containing one, two, or three integers separated by commas, then the first integer is the axis to stack the arrays along; the second is the minimum number of dimentions to force each entry into; the third allows you to control the shape of the resulting array.

If the first element in the index is "r" or "c", then a numpy **matrix** object is returned.

c_ works exactly like **r_**, but is column-oriented rather than row-oriented.

Example

np_tricks.py

```
import numpy as np

a1 = np.r_[np.array([1,2,3]), 0, 0, np.array([4,5,6])]
print a1
print

a2 = np.r_[-1:1:6j, [0]*3, 5, 6]
print a2
print

a = np.array([[0, 1, 2], [3, 4, 5]])
a3 = np.r_['-1', a, a]
print a3
print

a4 = np.r_['0,2', [1,2,3], [4,5,6]]
print a4
print

a5 = np.r_['0,2,0', [1,2,3], [4,5,6]]
print a5
print

a6 = np.r_['1,2,0', [1,2,3], [4,5,6]]
print a6
print

m = np.r_['r',[1,2,3], [4,5,6]]
print m
print type(m)
```

```
np_tricks.py
[1 2 3 0 0 4 5 6]

[-1.  -0.6 -0.2  0.2  0.6  1.   0.   0.   0.   5.   6. ]

[[0 1 2 0 1 2]
 [3 4 5 3 4 5]]

[[1 2 3]
 [4 5 6]]

[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]

[[1 4]
 [2 5]
 [3 6]]

[[1 2 3 4 5 6]]
```

```
<class 'numpy.matrixlib.defmatrix.matrix'
```

Matrices

- **Based on arrays**
- **Some operations are not element-wise**
- **Slicing a matrix always returns a matrix**
- **Use the A attribute to get at the underlying array**

NumPy provides the **matrix** type, which is based on the **array**. It is used for traditional matrix operations. The **A** attribute of a matrix is the array representation of that matrix.

Scalar multiplication of a matrix is the same as for an array, but multiplying one matrix times another follows a completely different algorithm.

Example

np_matrices.py

```
import numpy as np

m1 = np.matrix([[ 2,  4,  6],
        [10, 20, 30]])

m2 = np.matrix([[ 1, 15],
        [ 3, 25],
        [ 5, 35]])

print 'm1 =>\n', m1
print

print 'm2 =>\n', m2
print

print 'm1 * 3 =>\n', m1 * 3
print

print 'm1 * m2 =>\n', m1 * m2
print

print 'm1.A.transpose() =>\n', m1.A.transpose()
print

print 'm1.A.transpose() * m2.A =>\n', m1.A.transpose() * m2.A
```

np_matrices.py

```
m1 =>
[[ 2  4  6]
 [10 20 30]]

m2 =>
[[ 1 15]
 [ 3 25]
 [ 5 35]]

m1 * 3 =>
[[ 6 12 18]
 [30 60 90]]

m1 * m2 =>
[[  44  340]
 [ 220 1700]]

m1.A.transpose() =>
[[ 2 10]
 [ 4 20]
 [ 6 30]]

m1.A.transpose() * m2.A =>
[[   2  150]
 [  12  500]
 [  30 1050]]
```

Data Types

- **Data type is inferred from initialization data**
- **Can be specified with arange(), ones(), zeros(), etc.**

Numpy defines many different numeric data types. Integers can have different sizes and byte orders, and be either signed or unsigned. The data type is normally inferred from the initialization data. When using arange(), ones(), etc., to create arrays, the **dtype** parameter can be used to specify the data type.

Example

np_data_types.py

```
import numpy as np

r1 = np.arange(45)
r1.shape = (3,3,5)
print 'r1 =>\n', r1

r2 = np.arange(45,dtype=np.float)
r2.shape = (3,3,5)
print 'r2 =>\n', r2
```

```
np_data_types.py
r1 =>
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]
r2 =>
[[[  0.   1.   2.   3.   4.]
  [  5.   6.   7.   8.   9.]
  [ 10.  11.  12.  13.  14.]]

 [[ 15.  16.  17.  18.  19.]
  [ 20.  21.  22.  23.  24.]
  [ 25.  26.  27.  28.  29.]]

 [[ 30.  31.  32.  33.  34.]
  [ 35.  36.  37.  38.  39.]
  [ 40.  41.  42.  43.  44.]]]
```

Numpy Example List with Doc

- **Over 200 functions**

- Examples with doc strings

Numpy has over 200 functions. To get more information, check out the NumPy Example List with Doc at http://wiki.scipy.org/Numpy_Example_List_With_Doc. This page has all 217 (as of Spring 2014) of the Numpy functions and operators, with examples, and interleaved with their docstrings. It is a great resource for getting up to speed with Numpy.

Chapter 16 Exercises

Exercise 16-1

Starting with the file **big_arrays.py**, convert the Python list values into a numpy array.

Make a copy of the array named **values_x_3** with all values multiplied by 3.

Print out **values_x_3**

Exercise 16-2

Using arange(), create an array of 35 elements.

Reshape the arrray to be 5 x 7 and print it out.

Reshape the array to be 7 x 5 and print it out.

Exercise 16-3

Using **linspace()**, create an array of 500 elements evenly spaced between 100 and 200.

Reshape the array into 5 x 10 x 10.

Multiply every element by .5

Print the result.

Chapter 17

scipy

Chapter 17 Objectives

- **Understand the motivation for scipy**
- **See what the scipy module provides**
- **Import scipy and friends using standard abbreviations**
- **Learn which functions are aliased from numpy**
- **Use the scipy documentation commands**
- **Extend scipy with C/C++ via Weave**

About scipy

- **Part of the "Python Scientific Stack"**
- **Often used with matplotlib**
- **Many mathematical and statistical algorithms**
- **Includes numpy "under the hood"**
- **Can be used with iPython**
- **Very large collection of routines and subpackages**

scipy is a collection of modules and submodules for doing scientific (mostly numerical) analysis.

The scipy module itself acts as an umbrella module, or repository, for many useful submodules.

Although it can be used alone, **numpy** is part of **scipy**, and many useful **numpy** functions are aliased into the **scipy** namespace.

In addition, many common functions from **scipy's** dozens of submodules have been aliased to the **scipy** namespace.

Polynomials

- **Create a** poly1d **object**
- **Represented in either of two ways**
- **list of coefficients (1st element is coefficient of highest power)**
- **list of roots**
- **Call polynomial with value to solve for**
- r **attribute represents list of roots**

Polynomials can be represented in scipy in two ways using numpy's **poly1d**() method, which takes a list of coefficients; the other is to just provide a list of coefficients, where the first element is the coefficient of the highest power.

The **poly1d**() method takes a list of coefficients (or roots) and returns a **poly1d** object.

Treating the polynomial object like a string returns a text representation of the polynomial.

By default, pass an iterable of integers to **poly1d** which represent the coefficients. To specify roots, pass a second parameter to **poly1d** with a true value.

The variable used when displaying the polynomial is normally **x**. To use a different variable, add a third parameter with a string.

To solve for a specific value, call the polynomial with that value. The **r** property of the polynomial contains the roots.

You can use the addition, subtraction, division, multiplication, and exponentiation operators between polynomials and scalar values.

poly1d is automatically imported to **scipy**'s namespace as well

Example

sp_polynomials.py

```
import scipy as sp

# 2,1,4 are coefficients
p1 = sp.poly1d([2, 1, 4])
print p1
print
# evaluate for x = .75
print p1(.75)
# get the roots
print p1.r
print

# 2,1,4 are roots
p2 = sp.poly1d([2, 1, -4], True)
print p2
print
# evaluate for x = .75
print p2(.75)
# get the roots
print p2.r
print


# 1,2,3 are coefficients, variable is 'm'
p3 = sp.poly1d([1, 2, 3], False, 'm')
print p3
print
# evaluate for m = 100
print p3(100)
# get the roots
print p3.r
print

# polynomial arithmetic
p4 = sp.poly1d([1,2])
p5 = sp.poly1d([3,4])
print p4
print
print p5
print
print p4 + p5
print
print p4 - p5
print
print p4 ** 3
print
```

sp_polynomials.py

```
   2
2 x + 1 x + 4

5.875
[-0.25+1.39194109j -0.25-1.39194109j]

   3     2
1 x + 1 x - 10 x + 8

1.484375
[-4.  2.  1.]

   2
1 m + 2 m + 3

10203
[-1.+1.41421356j -1.-1.41421356j]
1 x + 2


3 x + 4


4 x + 6


-2 x - 2

   3     2
1 x + 6 x + 12 x + 8
```

Vectorizing functions

- **Many functions "just work"**
- **sp.vectorize() allows function to be broadcast.**

As with **numpy**, most **scipy** functions will automatically be broadcast across any array to which they are applied. For user-defined functions that don't correctly broadcast, scipy provides the **vectorize** function. It takes a function which accepts one or more scalar values (float, integers, etc.) and returns a single scalar value.

Example

sp_vectorize.py

```
import numpy as np
import scipy as sp

def set_default(value, limit, default):
    if value > limit:
        value = default

    return value

raw_samples = np.array([5, 18, 36, 1000, 98, 2323])

try:
    print "Without sp.vectorize:"
    norm_samples = set_default(raw_samples, 100, 0)
except ValueError as err:
    print "Function failed:", err
else:
    print norm_samples
finally:
    print

set_default_vect = sp.vectorize(set_default)
try:
    print "With sp.vectorize:"
    norm_samples = set_default_vect(raw_samples, 100, 0)
except ValueError as err:
    print "Function failed:", err
else:
    print norm_samples
finally:
    print
```

sp_vectorize.py

```
Without sp.vectorize:
Function failed: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()

With sp.vectorize:
[ 5 18 36  0 98  0]
```

Doing Real Work

- **Some functions imported to scipy**
- **Hundreds more in subpackages**
- **Most functions have similar interfaces**
- **Use numpy plus scipy.*subpackage* routines as needed**

scipy's subpackages have hundreds of functions. For convenience, some of them have been imported into the scipy namespace. While numpy is mostly about arrays and matrices, there are some useful data handling functions as well.

Example

sp_functions.py

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

dt = np.dtype([('Month', 'int8'), ('Day', 'int8'), ('Year', 'int16'),
('Temp', 'float64')])
data = np.loadtxt('../DATA/weather/NYNEWYOR.txt',dtype=dt)

print data['Temp']

print "First plot:"

days = xrange(len(data['Temp']))
temps = data['Temp']

plt.plot(days, temps )
plt.show()

print "Second plot:"

# hmmm -- some anomalous data
# let's remove readings < -50, which seem to be default N/A values
normalized_data = data[ data['Temp'] > -50 ]
days = xrange(len(normalized_data))
temps = normalized_data['Temp']
plt.plot(days, temps)
plt.show()
```

```
sp_functions.py
<see plot>
```

SciPy Subpackages

- **cluster** -- Clustering algorithms
- **constants** -- Physical and mathematical constants
- **fftpack** --Fast Fourier Transform routines
- **integrate** -- Integration and ordinary differential equation solvers
- **interpolate** -- Interpolation and smoothing splines
- **io** -- Input and Output
- **linalg** -- Linear algebra
- **ndimage** -- N-dimensional image processing
- **odr** -- 0rthogonal distance regression
- **optimize** -- Optimization and root-finding routines
- **signal** -- Signal processing
- **sparse** -- Sparse matrices and associated routines
- **spatial** -- Spatial data structures and algorithms
- **special** -- Special functions
- **stats** -- Statistical distributions and functions
- **weave** -- C/C++ integration

Getting Help

- **sp.info(*topic)***
- **sp.source(*module-or-method)***

In addition to the normal Python and iPython help systems, scipy provides its own **info()** method, which displays extended help on any scipy method.

The **source()** method displays the source code of the specified module or method.

Weave

- **Extend scipy with C/C++**
- **Three approaches**
- **weave.inline()**
- **weave.blitz()**
- **weave.ext_tools()**

When there is not a fast numpy/scipy routine that implements a needed algorithm, you can write the algorithm in C yourself.

This may not help with code that already uses existing numpy/scipy functions, but if the code contains nested loops, the speedup can be significant.

Of course, extending scipy with C requires a C development package. This can be either **Microsoft Visual C++** or **MinGW** on Windows, and is typically **gcc** on other platforms.

weave.inline

The simplest approach is **weave.inline**. To use it, create a Python function that accepts the data to be processed. In the function, define a string containing the C code. Pass the code, followed by an iterable of the *names* (as strings) of the functions's parameters, to **weave.inline**. Your Python function must return the value returned by **inline()**.

weave.blitz

While numpy arrays are much faster than standard Python lists, you can speed up numpy array access by using **weave.blitz**. This takes a string representing numpy array operations and compiles the string into C++, in a similar way to **weave.inline**. In the same way that **weave.inline** speeds up normal Python list operations, **weave.blitz** speeds up NumPy array operations.

weave.ext_tools

The ext_tools module in the weave package allows you to programmatically create C extension modules by defining modules via calls to Python functions.

There are other ways to speed up Python that are not part of **scipy:**

ctypes is a module that allows direct import C/C++ shared libraries (DLLs) without writing any custom C/C++ code

cython is a version of Python that automatically pre-compiles se;ected Python code into C. It looks like Python, but has type declarations similar to those in C.

f2py is a utility for calling Fortran code from Python

Example

sp_weave_inline.py

```
from scipy import weave


def times_n(seq,n):

    code = r'''
#include <stdio.h>
int val;
for (int i = 0; i < seq.length(); i++) {
    val = py_to_int(PyList_GetItem(seq,i),"val");
    printf("%02d %d\n", i, val * n);
}
'''

    return weave.inline(code,['seq','n'])

times_n([1,5,9], 10)
times_n([2,4,6,8,10,12], 30)
```

sp_weave_inline.py

```
00 10
01 50
02 90
00 60
01 120
02 180
03 240
04 300
05 360
```

Chapter 18

A Tour of scipy subpackages

Chapter 18 Objectives

- **Quickly view the functions available in all important scipy subpackages**

cluster

- **scipy.cluster.vq – quantization and k-means**
- **whiten(), vq(), kmeans(), kmeans2()**
- **scipy.cluster.hierarchy – hiearchical and agglomerative clustering**
- **Many functions (see next page for chart)**

The cluster.vq subpackage provides functions that implement clustering algorithms for target detection, communications, compressions, and information theory, among other areas.

The cluster.hierarchy subpackage implements algorithms for hierarchical and agglomerative functions.

fcluster(Z, t[, criterion, depth, R, monocrit])	*Flat clusters from the hierarchical clustering defined by*
fclusterdata(X, t[, criterion, metric, ...])	*cluster observation data using a given metric.*
leaders(Z, T)	*(L, M) = leaders(Z, T):*

linkage(y[, method, metric])	Hierarchical/agglomerative clustering on the condensed distance matrix y.
single(y)	Performs single/min/nearest linkage on the condensed distance matrix y
complete(y)	Performs complete/max/farthest point linkage on a condensed distance matrix
average(y)	Performs average/UPGMA linkage on a condensed distance matrix
weighted(y)	Performs weighted/WPGMA linkage on the condensed distance matrix
centroid(y)	Performs centroid/UPGMC linkage. See linkage for more
median(y)	Performs median/WPGMC linkage. See linkage for more
ward(y)	Performs Ward's linkage on a condensed or redundant distance

cophenet(Z[, Y])	Calculates the cophenetic distances between each observation in
from_mlab_linkage(Z)	Converts a linkage matrix generated by MATLAB(TM) to a new
inconsistent(Z[, d])	Calculates inconsistency statistics on a linkage.
maxinconsts(Z, R)	Returns maxinconsistency coefficient for each non-singleton cluster and descendents.
maxdists(Z)	Returns the maximum distance between any non-singleton cluster.
maxRstat(Z, R, i)	Returns the maximum statistic for each non-singleton cluster and its descendents.
to_mlab_linkage(Z)	Converts a linkage matrix Z generated by the linkage function

constants

- **Contains many useful constants**
- **Also contains 2010 CODATA recommended values**
- **CODATA constants accessed by key**

The constants subpackage contains many standard constants, plus the 2010 CODATA recommended values for physical constants.

Example

sp_constants.py

```
from scipy import constants as K

print "pi: {0}".format(K.pi)
print "Planck: {0}".format(K.Planck)
print "c (speed of light): {0}".format(K.c)

print "natural unit of energy: {0}".format(K.value('natural unit of
energy'))
print "natural unit of energy (Unit): {0}".format(K.unit('natural unit
of energy'))
```

```
sp_constants.py
pi: 3.14159265359
Planck: 6.62606957e-34
c (speed of light): 299792458.0
natural unit of energy: 8.18710506e-14
natural unit of energy (Unit): J
```

fftpack

- **Provides FFTs**
- **5 groups of methods**
  - **FFTs**
  - **Differential and pseudo-differential operators**
  - **Helper functions**
  - **Convolutions (scipy.fftpack.convolve)**
  - **Other (scipy.fftpack._fftpack)**

As the name implies, fftpack provices functions to compute Fast Fourier Transforms. There are one, two, and multi-dimensional transforms and their inverses, as well as Discrete Cosine Transforms.

The second group of functions provides differential and pseudo-differential operations.

The remaining functions are helper functions, convolutions ,

integrate

- **Functions for computing integrals**

The scipy.integrate subpackage contains routines for computing integrals.

See scipy.special for orthogonal polynomials

quad(func, a, b[, args, full_output, ...])	Compute a definite integral.
dblquad(func, a, b, gfun, hfun[, args, ...])	Compute a double integral.
tplquad(func, a, b, gfun, hfun, qfun, rfun)	Compute a triple (definite) integral.
fixed_quad(func, a, b[, args, n])	w/ fixed-order Gaussian quadrature.
quadrature(func, a, b[, args, tol, rtol, ...])	w/ fixed-tolerance quadrature.
romberg(function, a, b[, args, tol, rtol, ...])	Romberg integration

trapz(y[, x, dx, axis]) Integrate along given axis using the composite trapezoidal rule.
cumtrapz(y[, x, dx, axis, initial]) Cumulatively integrate using composite trapezoidal rule
simps(y[, x, dx, axis, even]) Integrate y(x) samples along given axis and composite
romb(y[, dx, axis, show]) Romberg integration using samples of a function.

odeint(func, y0, t[, args, Dfun, col_deriv, ...]) Integrate a system of ordinary differential equations.
ode(f[, jac]) A generic interface class to numeric integrators.
complex_ode(f[, jac]) A wrapper of ode for complex systems.

interpolate

- **Large number of functions**
- **Splines, one- and multi-dimensional interpolation**
- **Lagrange and Taylor interpolators**
- **FITPACK and DFITPACK functions**

This sub-package contains spline functions and classes, one-dimensional and multi-dimensional (univariate and multivariate) interpolation classes, Lagrange and Taylor polynomial interpolators, plus wrappers for FITPACK and DFITPACK functions

See also:
scipy.ndimage.map_coordinates, scipy.ndimage.spline_filter, scipy.signal.resample, scipy.signal.bspline, scipy.signal.gauss_spline,scipy.signal.qspline1d, scipy.signal.cspline1d, scipy.signal.qspline1d_eval, scipy.signal.cspline1d_eval, scipy.signal.qspline2d, scipy.signal.cspline2d.

interp1d(x, y[, kind, axis, copy, ...]) Interpolate a 1-D function.

BarycentricInterpolator(xi[, yi]) The interpolating polynomial for a set of points

KroghInterpolator(xi, yi) The interpolating polynomial for a set of points

PiecewisePolynomial(xi, yi[, orders, direction]) Piecewise polynomial curve

pchip(x, y) PCHIP 1-d monotonic cubic interpolation

barycentric_interpolate(xi, yi, x) Convenience function for polynomial interpolation

krogh_interpolate(xi, yi, x[, der]) -- for polynomial interpolation.

piecewise_polynomial_interpolate(xi, yi, x) -- for piecewise polynomial interpolation

griddata(points, values, xi[, method, ...]) Interpolate unstructured N-dimensional data.

LinearNDInterpolator(points, values) Piecewise linear interpolant in N dimensions.

NearestNDInterpolator(points, values) Nearest-neighbour interpolation in N dimensions.

CloughTocher2DInterpolator(points, values[, tol]) Piecewise cubic, C1 smooth, curvature-minimizing interpolant in 2D.

Rbf(*args) A class for radial basis function approximation/interpolation of n-dimensional scattered data.

interp2d(x, y, z[, kind, copy, ...]) Interpolate over a 2-D grid.

RectBivariateSpline(x, y, z[, bbox, kx, ky, s]) Bivariate spline approximation

UnivariateSpline(x, y[, w, bbox, k, s]) One-dimensional smoothing spline\
InterpolatedUnivariateSpline(x, y[, w, bbox, k]) One-dimensional interpolating spline
LSQUnivariateSpline(x, y, t[, w, bbox, k]) 1D spline with explicit internal knots.

UnivariateSpline.__call__(x[, nu]) Evaluate spline at positions x.
UnivariateSpline.derivatives(x) Return all derivatives of the spline at the point x.
UnivariateSpline.integral(a, b) Return definite integral of spline
UnivariateSpline.roots() Return the zeros of the spline.
UnivariateSpline.get_coeffs() Return spline coefficients.
UnivariateSpline.get_knots() Return positions of (boundary and interior) knots
UnivariateSpline.get_residual() Return weighted sum of squared residuals
UnivariateSpline.set_smoothing_factor(s) Continue spline with given smoothing

splrep(x, y[, w, xb, xe, k, task, s, t, ...]) Find the B-spline representation of 1-D curve.
splprep(x[, w, u, ub, ue, k, task, s, t, ...]) Find the B-spline representation of N-D curve.
splev(x, tck[, der, ext]) Evaluate a B-spline or its derivatives.
splint(a, b, tck[, full_output]) Evaluate the definite integral of a B-spline.
sproot(tck[, mest]) Find the roots of a cubic B-spline.
spalde(x, tck) Evaluate all derivatives of a B-spline.
bisplrep(x, y, z[, w, xb, xe, yb, ye, kx, ...]) Find a bivariate B-spline of a surface.
bisplev(x, y, tck[, dx, dy]) Evaluate a bivariate B-spline and its derivatives.

RectBivariateSpline(x, y, z[, bbox, kx, ky, s] Bivariate spline approximation
RectSphereBivariateSpline(u, v, r[, s, ...]) Bivariate spline approximation on a sphere.
BivariateSpline Bivariate spline s(x,y) on the rectangle [xb,xe] x [yb, ye]
SmoothBivariateSpline(x, y, z[, w, bbox, ...]) Smooth bivariate spline approximation.
LSQBivariateSpline(x, y, z, tx, ty[, w, ...]) Weighted least-squares approximation.
bisplrep(x, y, z[, w, xb, xe, yb, ye, kx, ...]) Bivariate B-spline representation of a surface.
bisplev(x, y, tck[, dx, dy]) Evaluate a bivariate B-spline and its derivatives.

lagrange(x, w) Return a Lagrange interpolating polynomial.
approximate_taylor_polynomial(f, x, degree, ...) Estimate the Taylor polynomial

io

- **Read/write data**
- **Supports MATLAB, IDL, MM, WAV, ARFF, NETCDF**
- **For some formats, use appropriate subpackage:**
  - **scipy.io.wavfile**
  - **scipy.io.arff**
  - **scipy.io.netcdf**

The scipy.io subpackage allows the user to read data from, and write data to, file. Many formats are supported. You may, however, prefer to use **pandas** for working with large data files.

loadmat(file_name[, mdict, appendmat]) Load MATLAB file

savemat(file_name, mdict[, appendmat, ...]) Save a dictionary of names and arrays into a MATLAB-style .mat file.

readsav(file_name[, idict, python_dict, ...]) Read an IDL .sav file

mminfo(source) Queries the contents of the Matrix Market file 'filename'

mmread(source) Reads the contents of a Matrix Market file 'filename' into a matrix.

mmwrite(target, a[, comment, field, precision]) Writes to an MM formatted file.

read(file) Return the sample rate (in samples/sec) and data from a WAV file

write(filename, rate, data) Write a numpy array as a WAV file

loadarff(f) Read an arff file.

netcdf_file(filename[, mode, mmap, version]) A file object for NetCDF data.

netcdf_variable(data, typecode, size, shape, ...) A data object for the netcdf module.

linalg

- **Many routines for linear algebra**
- **Imperfect subset of numpy.linalg**

The scipy.linalg subpackage provides routines for solving and computing linear algebra functions. Many of the routines are imported from numpy, but some of the scipy routines might be a little different.

Basics

inv(a[, overwrite_a]) Compute the inverse of a matrix.

solve(a, b[, sym_pos, lower, overwrite_a, ...]) Solve the equation a x = b for x.

solve_banded((l, u), ab, b[, overwrite_ab, ...]) Solve assuming a is banded matrix.

solveh_banded(ab, b[, overwrite_ab, ...]) Solve equation a x = b.

solve_triangular(a, b[, trans, lower, ...]) Solve assuming a is a triangular matrix.

det(a[, overwrite_a]) Compute the determinant of a matrix

norm(a[, ord]) Matrix or vector norm.

lstsq(a, b[, cond, overwrite_a, overwrite_b]) Compute least-squares solution

pinv(a[, cond, rcond]) Compute the (Moore-Penrose) pseudo-inverse of a matrix.

pinv2(a[, cond, rcond]) Compute the (Moore-Penrose) pseudo-inverse of a matrix.

kron(a, b) Kronecker product of a and b.

tril(m[, k]) Make a copy of a matrix with elements above the k-th diagonal zeroed.

triu(m[, k]) Make a copy of a matrix with elements below the k-th diagonal zeroed.

Eigenvalue Problems

eig(a[, b, left, right, overwrite_a, ...]) Solve an ordinary or generalized eigenvalue problem of a square matrix.

eigvals(a[, b, overwrite_a]) Compute eigenvalues from an ordinary or generalized eigenvalue problem.

eigh(a[, b, lower, eigvals_only, ...]) Solve an ordinary or generalized eigenvalue problem for a complex

eigvalsh(a[, b, lower, overwrite_a, ...]) Solve an ordinary or generalized eigenvalue problem for a complex

eig_banded(a_band[, lower, eigvals_only, ...]) Solve real symmetric or complex hermitian band matrix eigenvalue problem.

eigvals_banded(a_band[, lower, ...]) Solve real symmetric or complex hermitian band matrix eigenvalue problem.

Decompositions

lu(a[, permute_l, overwrite_a]) Compute pivoted LU decompostion of a matrix.

lu_factor(a[, overwrite_a]) Compute pivoted LU decomposition of a matrix.

lu_solve((lu, piv), b[, trans, overwrite_b]) Solve an equation system, a x = b, given the LU factorization of a

svd(a[, full_matrices, compute_uv, overwrite_a]) Singular Value Decomposition.

svdvals(a[, overwrite_a]) Compute singular values of a matrix.

diagsvd(s, M, N) Construct the sigma matrix in SVD from singular values and size M, N.

orth(A) Construct an orthonormal basis for the range of A using SVD

cholesky(a[, lower, overwrite_a]) Compute the Cholesky decomposition of a matrix.

cholesky_banded(ab[, overwrite_ab, lower]) Cholesky decompose a banded Hermitian positive-definite matrix

cho_factor(a[, lower, overwrite_a]) Compute the Cholesky decomposition of a matrix, to use in cho_solve

cho_solve((c, lower), b[, overwrite_b]) Solve the linear equations A x = b, given the Cholesky factorization of A.

cho_solve_banded((cb, lower), b[, overwrite_b]) Solve the linear equations A x = b, given the Cholesky factorization of A.

qr(a[, overwrite_a, lwork, mode, pivoting]) Compute QR decomposition of a matrix.

qr_multiply(a, c[, mode, pivoting, ...]) Calculate the QR decomposition and multiply Q with a matrix.

qz(A, B[, output, lwork, sort, overwrite_a, ...]) QZ decompostion for generalized eigenvalues of a pair of matrices.

schur(a[, output, lwork, overwrite_a, sort]) Compute Schur decomposition of a matrix.

rsf2csf(T, Z) Convert real Schur form to complex Schur form.

hessenberg(a[, calc_q, overwrite_a]) Compute Hessenberg form of a matrix.

Matrix Functions

expm(A[, q]) Compute the matrix exponential using Pade approximation.

expm2(A) Compute the matrix exponential using eigenvalue decomposition.

expm3(A[, q]) Compute the matrix exponential using Taylor series.

logm(A[, disp]) Compute matrix logarithm.

cosm(A) Compute the matrix cosine.

sinm(A) Compute the matrix sine.

tanm(A) Compute the matrix tangent.

coshm(A) Compute the hyperbolic matrix cosine.

sinhm(A) Compute the hyperbolic matrix sine.

tanhm(A) Compute the hyperbolic matrix tangent.

signm(a[, disp]) Matrix sign function.

sqrtm(A[, disp]) Matrix square root.

funm(A, func[, disp]) Evaluate a matrix function specified by a callable.

Matrix Equation Solvers

solve_sylvester(a, b, q) Computes a solution (X) to the Sylvester equation (AX + XB = Q).

solve_continuous_are(a, b, q, r) Solves the continuous algebraic Riccati equation, or CARE, defined

solve_discrete_are(a, b, q, r) Solves the disctrete algebraic Riccati equation, or DARE, defined as

solve_discrete_lyapunov(a, q) Solves the Discrete Lyapunov Equation (A'XA-X=-Q) directly.

solve_lyapunov(a, q) Solves the continuous Lyapunov equation (AX + XA^H = Q) given the values

Special Matrices

block_diag(*arrs) Create a block diagonal matrix from provided arrays.

circulant(c) Construct a circulant matrix.

companion(a) Create a companion matrix.

hadamard(n[, dtype]) Construct a Hadamard matrix.

hankel(c[, r]) Construct a Hankel matrix.

hilbert(n) Create a Hilbert matrix of order n.

invhilbert(n[, exact]) Compute the inverse of the Hilbert matrix of order n.

leslie(f, s) Create a Leslie matrix.

pascal(n[, kind, exact]) Returns the n x n Pascal matrix.

toeplitz(c[, r]) Construct a Toeplitz matrix.

tri(N[, M, k, dtype]) Construct (N, M) matrix filled with ones at and below the k-th diagonal.

ndimage

- **Many functions for N-dimensional image processing**

The scipy.ndimage subpackage provides numerous routines for image processing. For more image processing, see the Python Imaging Library (pil, imported as **Image)**

Filters

convolve(input, weights[, output, mode, ...]) n-D convolution.
convolve1d(input, weights[, axis, output, ...]) Calculate a 1-D convolution
correlate(input, weights[, output, mode, ...]) Multi-dimensional correlation.
correlate1d(input, weights[, axis, output, ...]) Calculate a 1-D correlatio along given axis.
gaussian_filter(input, sigma[, order, ...]) Multi-dimensional Gaussian filter.
gaussian_filter1d(input, sigma[, axis, ...]) One-dimensional Gaussian filter.
gaussian_gradient_magnitude(input, sigma[, ...]) Calculate n-D gaussian gradient
gaussian_laplace(input, sigma[, output, ...]) Calculate n-D laplace filter
generic_filter(input, function[, size, ...]) Calculates a n-D filter using the given function.
generic_filter1d(input, function, filter_size) Calculate 1-D filter along the given axis.
generic_gradient_magnitude(input, derivative) Gradient magnitude using given function
generic_laplace(input, derivative2[, ...]) n-D laplace filter using second derivative
laplace(input[, output, mode, cval]) n-D laplace filter using differences.
maximum_filter(input[, size, footprint, ...]) 1-D maximum filter.
maximum_filter1d(input, size[, axis, ...]) 1-D maximum filter along the given axis.
median_filter(input[, size, footprint, ...]) n-D median filter.
minimum_filter(input[, size, footprint, ...]) n-D minimum filter.
minimum_filter1d(input, size[, axis, ...]) 1-D minimum filter along the given axis.
percentile_filter(input, percentile[, size, ...]) n-D percentile filter.
prewitt(input[, axis, output, mode, cval]) Calculate a Prewitt filter.
rank_filter(input, rank[, size, footprint, ...]) Calculates a multi-dimensional rank filter.
sobel(input[, axis, output, mode, cval]) Calculate a Sobel filter.
uniform_filter(input[, size, output, mode, ...]) Multi-dimensional uniform filter.
uniform_filter1d(input, size[, axis, ...]) Calculate a one-dimensional uniform filter along the given axis.

Fourier filters

fourier_ellipsoid(input, size[, n, axis, output]) n-D ellipsoid fourier filter.
fourier_gaussian(input, sigma[, n, axis, output]) n-D Gaussian fourier filter.
fourier_shift(input, shift[, n, axis, output]) n-D fourier shift filter.
fourier_uniform(input, size[, n, axis, output]) n-D uniform fourier filter.

Image nterpolation

affine_transform(input, matrix[, offset, ...]) Apply an affine transformation.
geometric_transform(input, mapping[, ...]) Apply an arbritrary geometric transform.
map_coordinates(input, coordinates[, ...]) Map to new coordinates by interpolation.
rotate(input, angle[, axes, reshape, ...]) Rotate an array.
shift(input, shift[, output, order, mode, ...]) Shift an array.
spline_filter(input[, order, output]) Multi-dimensional spline filter.
spline_filter1d(input[, order, axis, output]) 1-D spline filter along the given axis.
zoom(input, zoom[, output, order, mode, ...]) Zoom an array.

Measurements

center_of_mass(input[, labels, index]) Calculate the center of mass
extrema(input[, labels, index]) Calculate min/max of the values of an array
find_objects(input[, max_label]) Find objects in a labeled array.
histogram(input, min, max, bins[, labels, index]) Histogram of the values of an array
label(input[, structure, output]) Label features in an array.
maximum(input[, labels, index]) Maximum of the values of an array over labeled regions.
maximum_position(input[, labels, index]) Positions of the maxima
mean(input[, labels, index]) Calculate the mean of the values of an array at labels.
minimum(input[, labels, index]) Calculate the minimum of the values of an array
minimum_position(input[, labels, index]) Find the positions of the minimums
standard_deviation(input[, labels, index]) Calculate the standard deviation
sum(input[, labels, index]) Calculate the sum of the values of the array.
variance(input[, labels, index]) Calculate the variance of the values of an n-D image
watershed_ift(input, markers[, structure, ...]) Apply watershed from markers

Morphology

binary_closing(input[, structure, ...]) n-D binary closing with given structuring element.
binary_dilation(input[, structure, ...]) n-D binary dilation w/ given structuring element.
binary_erosion(input[, structure, ...]) n-D binary erosion a given structuring element.
binary_fill_holes(input[, structure, ...]) Fill the holes in binary objects.
binary_hit_or_miss(input[, structure1, ...]) n-D binary hit-or-miss transform.
binary_opening(input[, structure, ...]) n-D binary opening w/ given structuring element.
binary_propagation(input[, structure, mask, ...]) n-D binary propagation
black_tophat(input[, size, footprint, ...]) n-D black tophat filter.
distance_transform_bf(input[, metric, ...]) Distance transform function by brute force
distance_transform_cdt(input[, metric, ...]) Distance transform for chamfer transforms.
distance_transform_edt(input[, sampling, ...]) Exact euclidean distance transform.
generate_binary_structure(rank, connectivity) Generate binary structure for binary morphological operations.
grey_closing(input[, size, footprint, ...]) n-D greyscale closing.
grey_dilation(input[, size, footprint, ...]) Greyscale dilationw/ structure or footprint
grey_erosion(input[, size, footprint, ...]) Greyscale erosion w/ structure or footprint
grey_opening(input[, size, footprint, ...]) n-D greyscale opening.
iterate_structure(structure, iterations[, ...]) Iterate a structure by dilating it with itself.
morphological_gradient(input[, size, ...]) n-D morphological gradient.
morphological_laplace(input[, size, ...]) n-D morphological laplace.
white_tophat(input[, size, footprint, ...]) n-D white tophat filter.

Utility

imread(fname[, flatten]) Load an image from file.

odr

- **Weighted orthogonal distance regression**
- **Minimizes sum of squared weighted distances**
- **Solve nonlinear least squares**
- **Uses Levenberg-Marquardt algorithm**

The scipy.odr subpackage provides both functions and a class to perform weighted orthogonal distance regression. This means determining the values to minimize the sum of squared, weighted orthogonal distances from a curve or surface.

The package also has functions to solve nonlinear ordinary least squares.

There are strong error checking and reporting features as well.

Function

odr(fcn, beta0, y, x[, we, wd, fjacb, ...])

Classes

ODR(data, model[, beta0, delta0, ifixb, ...])

The ODR class gathers all information and coordinates the running of the main fitting routine.

Data(x[, y, we, wd, fix, meta]) The Data class stores the data to fit.

Model(fcn[, fjacb, fjacd, extra_args, ...]) The Model class stores information about the function you wish to fit.

Output(output) The Output class stores the output of an ODR run.

RealData(x[, y, sx, sy, covx, covy, fix, meta]) The RealData class stores the weightings as actual standard deviations

Exceptions

odr_error

odr_stop

For details on this subpackage, see the User's Reference Guide at http://docs.scipy.org/doc/external/odrpack_guide.pdf

optimize

- **Many functions for optimization and root finding**
- **Four main divisions**
  - **Optimization**
  - **Fitting**
  - **Root Finding**
  - **General utility**

The scipy.optimize module contains functions that implement common optimization algorithms.

Optimization

General-purpose

minimize(fun, x0[, args, method, jac, hess, ...]) Minimize scalar function 1+ variables.
fmin(func, x0[, args, xtol, ftol, maxiter, ...]) Minimize w/ the downhill simplex algorithm.
fmin_powell(func, x0[, args, xtol, ftol, ...]) Minimize w/ modified Powell's method
fmin_cg(f, x0[, fprime, args, gtol, norm, ...]) Minimize w/ nonlinear conjugate gradient
fmin_bfgs(f, x0[, fprime, args, gtol, norm, ...]) Minimize with BFGS
fmin_ncg(f, x0, fprime[, fhess_p, fhess, ...]) Unconstrained minimization w/ Newton-CG
leastsq(func, x0[, args, Dfun, full_output, ...]) Minimize the sum of squares of equations.

Constrained (multivariate)

fmin_l_bfgs_b(func, x0[, fprime, args, ...]) Minimize function with L-BFGS-B algorithm.
fmin_tnc(func, x0[, fprime, args, ...]) Minimize function with variables subject to bounds
fmin_cobyla(func, x0, cons[, args, ...]) Minimize function w/ Constrained Optimization
fmin_slsqp(func, x0[, eqcons, f_eqcons, ...]) Minimize w/ Sequential Least SQuares
nnls(A, b) Solve argmin_x || Ax - b ||_2 for x>=0. *This is a wrapper*

Global

anneal(func, x0[, args, schedule, ...]) Minimize a function using simulated annealing.
brute(func, ranges[, args, Ns, full_output, ...]) Minimize by brute force.

Scalar function minimizers

minimize_scalar(fun[, bracket, bounds, ...]) Minimization scalar function of one variable.
fminbound(func, x1, x2[, args, xtol, ...]) Bounded minimization for scalar functions.
brent(func[, args, brack, tol, full_output, ...]) Minimize to a fractional precision of tol.
golden(func[, args, brack, tol, full_output]) Minimize to a fractional precision of tol.
bracket(func[, xa, xb, args, grow_limit, ...]) Bracket the minimum of the function.

Rosenbrock function

rosen(x) The Rosenbrock function.
rosen_der(x) The derivative (i.e. gradient) of the Rosenbrock function
rosen_hess(x) The Hessian matrix of the Rosenbrock function.
rosen_hess_prod(x, p) Product of Hessian matrix of Rosenbrock function with a vector.

Fitting

curve_fit(f, xdata, ydata[, p0, sigma]) Use non-linear least squares to fit function to data.

Root finding

Scalar functions

brentq(f, a, b[, args, xtol, rtol, maxiter, ...]) Find a root of a function in given interval.
brenth(f, a, b[, args, xtol, rtol, maxiter, ...]) Find root of f in [a,b].
ridder(f, a, b[, args, xtol, rtol, maxiter, ...]) Find a root of a function in an interval.
bisect(f, a, b[, args, xtol, rtol, maxiter, ...]) Find root of f in [a,b].
newton(func, x0[, fprime, args, tol, ...]) Find a zero using Newton-Raphson or secant
fixed_point(func, x0[, args, xtol, maxiter]) Find point where func(x) == x

Multidimensional

root(fun, x0[, args, method, jac, tol, ...]) Find a root of a vector function.

fsolve(func, x0[, args, fprime, ...]) Find the roots of a function.

broyden1(F, xin[, iter, alpha, ...]) Find root using Broyden's first Jacobian approximation.

broyden2(F, xin[, iter, alpha, ...]) Find a root using Broyden's second Jacobian approx.

newton_krylov(F, xin[, iter, rdiff, method, ...]) Find root using Krylov approximation

anderson(F, xin[, iter, alpha, w0, M, ...]) Find a root using (extended) Anderson mixing.

excitingmixing(F, xin[, iter, alpha, ...]) Find a root w/ tuned diagonal Jacobian approx.

linearmixing(F, xin[, iter, alpha, verbose, ...]) Find a root using a scalar Jacobian approx.

diagbroyden(F, xin[, iter, alpha, verbose, ...]) Find a root using diagonal Broyden Jacobian approximation.

Utility Functions¶

line_search(f, myfprime, xk, pk[, gfk, ...]) Find alpha that satisfies strong Wolfe conditions.

check_grad(func, grad, x0, *args) Check the correctness of a gradient function by comparing it against a (forward) finite-difference approximation of the gradient.

show_options(solver[, method]) Show documentation for additional options of optimization solvers.

Signal

- **Signal processing functions**
- **In SciPy, signal represented as array numbers**
- **Signal data can be real or complex**
- **Contains 100 functions!**

The scipy.signal subpackage contains around 100 functions for signal processing. Most of these are filtering functions , plus some B-spline functions similar to algorithms in the scipy.optimize package.

In SciPy is an array of real or complex numbers. The B-spline routines only work with equally spaced data.

Sparse

- **Implements ARPACK package from Fortran**
- **Use sparse.linalg subpackage bq**
- **Can compute**
  - **Real or complex nonsymmetric square matrices**
  - **Real-symmetric or complex-hermitian matrices**

ARPACK is a Fortran package which provides routines for quickly finding a few eigenvalues/eigenvectors of large sparse matrices. In order to find these solutions, it requires only left-multiplication by the matrix in question. This operation is performed through a *reverse-communication* interface. The result of this structure is that ARPACK is able to find eigenvalues and eigenvectors of any linear function mapping a vector to a vector.

All of the functionality provided in ARPACK is contained within the two high-level interfaces `scipy.sparse.linalg.eigs` and `scipy.sparse.linalg.eigsh`. `eigs`provides interfaces to find the eigenvalues/vectors of real or complex nonsymmetric square matrices, while `eigsh` provides interfaces for real-symmetric or complex-hermitian matrices.

spatial

- **Provides spatial algorithms**
  - **Nearest-neighbor problems (KDTree)**
  - **Delaunay triangulation**

The scipy.spatial package provides spatial algorithms. In particular, it provides a KDTree class for solving nearest-neighbor problems.

special

- **Miscellaneous functions**
  - **Airy**
  - **Elliptic**
  - **Bessel**
  - **Struve**
  - **Raw statistics**
  - **Gamma**
  - **Error functions and Fresnel integrals**
  - **Legendre**
  - **Orthogonal polynomials**
  - **Hypergeometrics**
  - **Parabolic cylinders**
  - **Mathieu and related**
  - **Spheroidal wave**
  - **Kelvin**
  - ***and other misc convenience functions***

The scipy.special subpackage contains a large number of miscellaneous functions in various disciplines.

Stats

- **Huge number of probability and stats classes**
- **Classes inherit methods from one of:**
  - **rv_continuous**
  - **rv_discrete**
- **Corresponding functions for masked arrays**

The scipy.stats subpackage is one of the most important, as it contains routines for calculating many types of statistics. Once set of functions is for continuous distributions, and the other is for discrete distributions.

The continuous functions return an object that inherits from the base class rv_continuous; the discrete functions return an object based on rv_discrete. Thus each returned object implements a consistent set of methods.

Chapter 19

pandas

Chapter 19 Objectives

- **Understand what the pandas module provides**
- **Load data from CSV and other files**
- **Access data tables**
- **Extract rows and columns using conditions**
- **Calculate statistics for rows or columns**

About pandas

- **Reads data from file, database, or other sources**
- **Deals with real-life issues such as invalid data**
- **Powerful selecting and indexing tools**
- **Builtin statistical functions**
- **Munge, clean, analyze, and model data**
- **Works with** numpy **and** matplotlib

pandas is a package designed to make it easy to get, organize, and analyze large datasets. Its strengths lie in its ability to read from many different data sources, and to deal with real-life issues, such as missing, incomplete, or invalid data.

pandas also contains functions for calculating means, sums and other kinds of analysis.

For selecting desired data, **pandas** has many ways to select and filter rows and columns.

It is easy to integrate **pandas** with numpy, matplotlib, and other scientific packages.

While **pandas** can handle one, two, three, or higher dimensional data, it is generally used with two-dimensional (row/column) data, which can be visualized like a spreadsheet.

pandas provides powerful Split-Apply-Combine operations -- **groupby** enables transformations, aggregations, and easy-access plotting functions. It is easy to emulate R's **plyr** package via pandas.

pandas gets its name from **pan**el **da**ta **s**ystem

Pandas architecture

- **pandas.core.frame**
- **Two main structures: Series and DataFrame**
- **Series – one-dimensional**
- **DataFrame – two-dimensional**

The two main data stuctures in pandas are the Series and the DataFrame. A series is a one-dimensional indexed list of values, something like an ordered dictionary. A DataFrame is is a two-dimensional grid, with both row and column indices (like the rows and columns of a spreadsheet, but more flexible).

You can specify the indices, or pandas will use successive integers. Each row or column of a DataFrame is a Series.

The third structure in pandas is a Panel, which is more more or less a collection of DataFrames, and describes three-dimensional (or higher) data.

Series

- **Indexed list of values**
- **Similar to a dictionary, but ordered**
- **Can get sum(), mean(), etc.**
- **Use index to get individual values**
- **Indices are *not* positional**

A Series is an indexed sequence of values. Each item in the sequence has an index. The default index is a set of increasing integer values, but any set of values can be used.

For example, you can create a series with the values 5, 10, and 15 as follows:

```
s1 = pd.Series([5,10,15])
```

This will create a Series indexed by [0, 1, 2]. To provide index values, add a second list:

```
s2 = pd.Series([5,10,15], ['a','b','c'])
```

This specifies the indices as 'a', 'b', and 'c'.

You can also create a Series from a dictionary. **pandas** will put the index values in order:

```
s3 = pd.Series({'b':10, 'a':5, 'c':15})
```

There are many methods that can be called on a Series, and Series can be indexed in many flexible ways.

Example

pandas_create_series.py

```
from pandas.core.frame import Series

s1 = Series([5,10,15])
print s1, "\n"
print "s1[0]:",s1[0], "\n"
print '-' * 60

s2 = Series([5,10,15], ['a','b','c'])
print s2, "\n"
print "s2['a']:", s2['a']
print '-' * 60

s3 = Series({'b':10, 'a':5, 'c':15})
print s3, "\n"
print "s3.sum(), s3.mean():", s3.sum(), s3.mean()
print '-' * 60
```

```
pandas_create_series_py
0     5
1    10
2    15

s1[0]: 5

------------------------------------------------------------
a     5
b    10
c    15

s2['a']: 5
------------------------------------------------------------
a     5
b    10
c    15

s3.sum(), s3.mean(): 30 10.0
------------------------------------------------------------
```

DataFrames

- **Two-dimensional grid of values**
- **Row and column labels (indices)**
- **Rich set of methods**
- **Powerful indexing**

A DataFrame is the workhorse of Pandas. It represents a two-dimensional grid of values, containing indexed rows and columns, something like a spreadsheet.

There are many ways to create a DataFrame. They can be modified to add or remove rows/columns. Missing or invalid data can be eliminated or normalized.

DataFrames can be initialized from many kinds of data. See the table on the next page for a list of possibilities.

The panda DataFrame is modeled after R's data.frame

DataFrame Initializers	
Initializer	**Description**
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

Example

pd_simple_dataframe.py

```
from pandas.core.frame import Series, DataFrame

cols = ['alpha','beta','gamma','delta','epsilon']
index = ['a','b','c','d','e','f']
values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print 'values:', values, '\n\n'

df = DataFrame(values, index=index, columns=cols)
print 'DataFrame df:\n', df, '\n'
```

```
pd_simple_dataframe.py
values: [[100, 110, 120, 130, 140], [200, 210, 220, 230, 240],
[300, 310, 320, 330, 340], [400, 410, 420, 430, 440], [500, 510,
520, 530, 540], [600, 610, 620, 630, 640]]


DataFrame df:
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640
```

Data alignment

- **pandas will auto-align data by rows and columns**
- **Non-overlapping data will be set as NaN**

When two dataframes are combined, columns and indices are aligned.

The result is the union of matching rows and columns. Where data doesn't exist in one or the other dataframe, it is set to NaN.

A default value can be specified for the overlapping cells when combining dataframes with methods such as add() or mul().

Use the **fill_value** parameter to set a default for missing values.

Example

panda_alignment.py

```
import numpy as np
from pandas.core.frame import DataFrame
from py4sci_util import print_header

dataset1 = np.arange(9.).reshape((3,3))

df1 = DataFrame(
    dataset1,
    columns = ['apple', 'banana', 'mango'],
    index = ['orange', 'purple', 'blue']
)

dataset2 = np.arange(12.).reshape((4,3))

df2 = DataFrame(
    dataset2,
    columns = ['apple', 'banana', 'kiwi'],
    index = ['orange', 'purple', 'blue', 'brown']
)

print_header('df1')
print df1
print

print_header('df2')
print df2
print

print_header('df1 + df2')
print df1 + df2

print_header('df1.add(df2, fill_value=0)')
print df1.add(df2, fill_value=0)
```

panda_alignment.py

```
================================================
= df1 =
================================================
        apple  banana  mango
orange      0       1      2
purple      3       4      5
blue        6       7      8

================================================
= df2 =
================================================
        apple  banana  kiwi
orange      0       1     2
purple      3       4     5
blue        6       7     8
brown       9      10    11

================================================
= df1 + df2 =
================================================
        apple  banana  kiwi  mango
blue       12      14   NaN    NaN
brown     NaN     NaN   NaN    NaN
orange      0       2   NaN    NaN
purple      6       8   NaN    NaN
================================================
= df1.add(df2, fill_value=0) =
================================================
        apple  banana  kiwi  mango
```

```
blue        12       14       8        8
brown        9       10      11      NaN
orange       0        2       2        2
purple       6        8       5        5
```

Index objects

- **Used to index Series or DataFrames**
- **index = pandas.core.frame.Index(*sequence*)**
- **Can be named**

An index object is a kind of ordered set that is used to access rows or columns in a dataset. As shown earlier, indexes can be specified as lists or other sequences when creating a Series or DataFrame.

You can create an index object and then create a Series or a DataFrame using the index object. Index objects can be named, either something obvious like 'rows' or columns', or more appropriate to the specific type of of data being indexed.

Remember that index objects act like sets, so the main operations on them are unions, intersections, or differences.

You can replace an existing index on a DataFrame with the **set_index()** method.

Example
pandas_index_objects.py

```
from pandas.core.frame import Index, Series, DataFrame

index1 = Index(['a','b','c'], name='letters')
index2 = Index(['b','a','c'])
index3 = Index(['b','c','d'])
index4 = Index(['red','blue','green'], name='colors')


print index1
print index2
print index3
print

# these are the same
print index2 & index3
print index2.intersection(index3)
print

# these are the same
print index2 | index3
print index2.union(index3)
print

print index1 - index3
print

series1 = Series([10,20,30], index=index1)
print series1
print

dataframe1 = DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index1,
columns=index4)
print dataframe1
print

dataframe2 = DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index4,
columns=index1)
print dataframe2
print
```

pandas_index_objects.py

```
Index([a, b, c], dtype=object)
Index([b, a, c], dtype=object)
Index([b, c, d], dtype=object)

Index([b, c], dtype=object)
Index([b, c], dtype=object)

Index([a, b, c, d], dtype=object)
Index([a, b, c, d], dtype=object)

Index([a], dtype=object)

letters
a          10
b          20
c          30

colors   red  blue  green
letters
a          1     2      3
b          4     5      6
c          7     8      9

letters  a  b  c
colors
red      1  2  3
blue     4  5  6
green    7  8  9
```

Basic Indexing

- **Similar to normal Python or numpy**
- **Slices select rows**

One of the real strengths of **pandas** is the ability to easily select desired rows and columns. This can be done with simple subscripting, like normal Python, or extended subscripting, similar to **numpy**. In addition, **pandas** has special methods and attributes for selecting data.

For selecting columns, use the column name as the subscript value. This selects the entire column. To select multiple columns, use a sequence (list, tuple, etc.) of column names.

For selecting rows, use slice notation. This may not map to similar tasks in normal python. That is, **dataframe[x:y]** selects rows x through y, but **dataframe[x]** selects *column* x.

Example
pandas_selecting.py

```
from pandas.core.frame import DataFrame

cols = ['alpha','beta','gamma','delta','epsilon']
index = ['a','b','c','d','e','f']
values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = DataFrame(values, index=index, columns=cols)
print 'DataFrame df:\n', df, '\n'
print

# select a column
print "df['alpha']:"
print df['alpha']
print

# same as above
print "df.alpha:"
print df.alpha
print

# slice rows
print "df['b':'e']"
print df['b':'e']
print

# slice columns
print "df[['alpha','epsilon','beta']]"
print df[['alpha','epsilon','beta']]
print
```

pandas_selecting.py

```
DataFrame df:
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640


df['alpha']:
a    100
b    200
c    300
d    400
e    500
f    600
Name: alpha

df.alpha:
a    100
b    200
c    300
d    400
e    500
f    600
Name: alpha

df['b':'e']
   alpha  beta  gamma  delta  epsilon
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540

df[['alpha','epsilon','beta']]
   alpha  epsilon  beta
a    100      140   110
b    200      240   210
c    300      340   310
d    400      440   410
e    500      540   510
f    600      640   610
```

Broadcasting

- **Operation is applied across rows and columns**
- **Can be restricted to selected rows/columns**
- **Sometimes called vectorization**
- **Use apply() for more complex operations**

If you multiply a dataframe by some number, the operation is broadcast, or vectorized, across all values. This is true for all basic math operations.

The operation can be restricted to selected columns.

For more complex operations, the apply() method will apply a function that selects elements. You can use the name of an existing function, or supply a lambda (anonymous) function.

Example
pandas_broadcasting.py

```
import pandas as pd
from pandas import DataFrame, Series

cols = ['alpha','beta','gamma','delta','epsilon']
index = pd.date_range('2013-01-01 00:00:00',periods=6, freq='D')

print index, "\n"

values = [
    [100, 110, 120, 930, 140],
    [250, 210, 120, 130, 840],
    [300, 310, 520, 430, 340],
    [275, 410, 420, 330, 777],
    [300, 510, 120, 730, 540],
    [150, 610, 320, 690, 640],
]

ser1 = Series([.1, .2, .3, .4, .5])

df = DataFrame(values, index, cols)
print "Basic DataFrame:"
print df
print

print "Triple each column"
print df * 3
print

print "Date of max value for each column"
print df.apply(lambda x: x.index[x.argmax()])
print

print "Multiply column gamma by 1.5"
df['gamma'] *= 1.5
print df
print
```

pandas_broadcasting.py

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00, ..., 2013-01-06 00:00:00]
Length: 6, Freq: D, Timezone: None

Basic DataFrame:
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110    120    930      140
2013-01-02    250   210    120    130      840
2013-01-03    300   310    520    430      340
2013-01-04    275   410    420    330      777
2013-01-05    300   510    120    730      540
2013-01-06    150   610    320    690      640

Triple each column
            alpha  beta  gamma  delta  epsilon
2013-01-01    300   330    360   2790      420
2013-01-02    750   630    360    390     2520
2013-01-03    900   930   1560   1290     1020
2013-01-04    825  1230   1260    990     2331
2013-01-05    900  1530    360   2190     1620
2013-01-06    450  1830    960   2070     1920

Date of max value for each column
alpha      2013-01-03 00:00:00
beta       2013-01-06 00:00:00
gamma      2013-01-03 00:00:00
delta      2013-01-01 00:00:00
epsilon    2013-01-02 00:00:00

Multiply column gamma by 1.5
            alpha  beta  gamma  delta  epsilon
2013-01-01    100   110    180    930      140
2013-01-02    250   210    180    130      840
2013-01-03    300   310    780    430      340
2013-01-04    275   410    630    330      777
2013-01-05    300   510    180    730      540
2013-01-06    150   610    480    690      640
```

Removing entries

- **Remove rows or columns**
- **Use drop() method**

To remove columns or rows, use the drop() method, with the appropriate labels. Use axis=1 to drop columns, or axis=0 to drop rows.

Example

pandas_drop.py

```
from pandas.core.frame import DataFrame

cols = ['alpha','beta','gamma','delta','epsilon']
index = ['a','b','c','d','e','f']
values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print 'values:', values, '\n\n'

df = DataFrame(values, index=index, columns=cols)
print 'DataFrame df:\n', df, '\n'
print

df2 = df.drop(['beta','delta'], axis=1)
print "After dropping beta and delta:"
print df2
print

print "After dropping rows b, c, and e
df3 = df.drop(['b','c','e'])
print df3
```

pandas_drop.py

```
values: [[100, 110, 120, 130, 140], [200, 210, 220, 230, 240],
[300, 310, 320, 330, 340], [400, 410, 420, 430, 440], [500, 510,
520, 530, 540], [600, 610, 620, 630, 640]]


DataFrame df:
   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
b    200   210    220    230      240
c    300   310    320    330      340
d    400   410    420    430      440
e    500   510    520    530      540
f    600   610    620    630      640


After dropping beta and delta:
   alpha  gamma  epsilon
a    100    120      140
b    200    220      240
c    300    320      340
d    400    420      440
e    500    520      540
f    600    620      640

   alpha  beta  gamma  delta  epsilon
a    100   110    120    130      140
d    400   410    420    430      440
f    600   610    620    630      640
```

Time Series

- **Use time_series()**
- **Specify start/end time/date, number of periods, time units**
- **Useful as index to other data**
- **freq=***time_unit*
- **periods=***number_of_periods*

pandas provides a function time_series() to generate a list of timestamps. You can specify the start/end times as dates or dates/times, and the type of time units. Alternatively, you can specify a start date/time and the number of periods to create.

The frequency strings can have multiples – **5H** means every 5 hours, **3S** means every 3 seconds, etc.

Units for time_series() *freq* flag

Unit	Represents
M	Month
D	Day
H	Hour
T	Minute
S	Second

Example

pd_time_slices.py

```
import pandas as pd
import numpy as np

# every hour for 3 days
hourly = pd.date_range('1/1/2013 00:00:00','1/3/2013 23:59:59',
freq='H')
print "Number of periods: ",len(hourly)

# every second for 18 hours
seconds = pd.date_range('1/1/2013
12:00:00',freq='S',periods=(60*60*18))
print "Number of periods: ", len(seconds)
print "Last second: ",seconds[-1]


# every month for 1 year
monthly = pd.date_range('1/1/2013','12/31/2013', freq='M')
print "Number of periods: {0} Seventh element: {1}".format(
len(monthly),
monthly[6]
)

NUM_DATA_POINTS = 1441 # number of minutes in a day

# create range from starting point with specified number of points
# one day's worth of minutes
dates = pd.date_range('4/1/2013 00:00:00', periods=NUM_DATA_POINTS,
freq='T')

# a day's worth of data
data = np.random.random(NUM_DATA_POINTS)

# series indexed by minutes
series = pd.Series(data,index=dates)

# grab the half hour of data from 10:00 to 10:30
time_slice = series['4/1/2013 10:00:00':'4/1/2013 10:30:00']
print time_slice # 31 values
```

```
pd_time_slices.py
Number of periods:  72
Number of periods:  64800
Last second:  2013-01-02 05:59:59
Number of periods: 12 Seventh element: 2013-07-31 00:00:00
2013-04-01 10:00:00    0.791706
2013-04-01 10:01:00    0.393151
...
2013-04-01 10:29:00    0.208319
2013-04-01 10:30:00    0.244398
Freq: T
31
```

Methods and attributes for fetching data	
Method	**Description**
***DF*.columns**	Get or set column labels
***DF*.shape()** ***S*.shape()**	Get or set shape (length of each axis)
***DF*.head(n)** ***DF*.tail(n)**	Return n items (default 5) from beginning or end
***DF*.values** ***S*.values**	Get the actual values from a data structure
***DF*.loc[row_indexer, col_indexer]**	Multi-axis indexing by label (not by position)
***DF*.iloc[row_indexer, col_indexer]**	Multi-axis indexing by integer position (not by labels)
***DF*.ix[row_indexer, col_indexer]**	Multi-axis indexing using either label or position (uses labels, but falls back to position)

Methods for Computations/Descriptive Stats	
Method	**Returns**
abs()	absolute values
corr()	pairwise correlations
count()	number of values
cov()	Pairwise covariance
cumsum()	cumulative sums
cumprod()	cumulative products
cummin(), cummax()	cumulative minimum, maximum
kurt()	unbiased kurtosis
median()	median
min(), max()	minimum, maximum values
prod()	products
quantile()	values at given quantile
skew()	unbiased skewness
std()	standard deviation
var()	variance
Note: these methods return Series or DataFrames, as appropriate, and can be computed over rows (axis=0) or columns (axis=1). They generally skip NA/null values.	

Reading Data

- **Supports many data formats**
- **Reads headings to create column indexes**
- **Auto-creates indexes as needed**
- **Can used specified column as row index**

pandas support many different input formats. It will reading file headings and use them to create column indexes. By default, it will use integers for row indices, but you can specify a column to use as the index.

The read methods have many options for controlling and parsing input. For instance, if large integers in the file contain commas, the *thousands* options let you set the separator as comma (in the US), so it will ignore them.

Pandas I/O methods

Methods	Reads the following into a DataFrame
read_table()	Generic delimited file
read_csv()	CSV file
read_fwf()	File with fixed-width fields
read_clipboard()	Data from OS clipboard (passed to **read_table**())
read_excel()	Excel table
read_html()	HTML table
read_hdf()	HDF5 Store (using PyTables)
read_sql()	Result of SQL query

Example

pandas_read_csv.py

```
import pandas as pd
from py4sci_util import print_header

# data from
#
http://www.rita.dot.gov/bts/sites/rita.dot.gov.bts/files/publications/
# national_transportation_statistics/html/table_01_44.html

airports_df =
pd.read_csv('../DATA/airport_boardings.csv',thousands=',')

print_header("ENTIRE DATAFRAME")

print airports_df, "\n"

print_header("ONLY COLUMN 'CODE'")
print airports_df['Code'], "\n"

print_header("SELECTED COLUMNS WITH FILTERED ROWS")
columns_wanted = ['Code', '2001 Total', 'Airport']
sort_col = '2001 Total'
max_val = 10000000
selector = airports_df['2001 Total'] > max_val
print airports_df[selector]
[columns_wanted].sort(sort_col,ascending=False)


print_header("COLUMN TOTALS")
print airports_df[['2001 Total','2010 Total']].sum(), "\n"

print_header("'CODE' COLUMN SET AS INDEX")
airports_df.set_index('Code')
print airports_df
```

```
pandas_read_csv.py
================================================
= ENTIRE DATAFRAME =
================================================
                                          Airport Code  2001 Rank  2001 Total  2010 Rank
2010 Total  2011 Rank     Total  Percent change 2001-2011  Percent change 2010-2011
0   Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ATL          1    36384264          1
42655392          1  33034788                      -9.2                     -22.6
1          Chicago, IL (Chicago O'Hare International)  ORD          2    28626694          2
30033313          2  22367052                     -21.9                     -25.5
2        Dallas, TX (Dallas/Fort Worth International)  DFW          3    25198256          3
26785739          3  20430281                     -18.9                     -23.7
3                Denver, CO (Denver International)  DEN          6    16397390          4
24965553          4  19190341                      17.0                     -23.1
4         Los Angeles, CA (Los Angeles International)  LAX          4    22873307          5
22860849          5  18379418                     -19.6                     -19.6
...
49   2817684                     12.5                    -22.0
49      Indianapolis, IN (Indianapolis International)  IND         48     3410636         50
3716884         50   2750105                    -19.4                      -26.0

================================================
= ONLY COLUMN 'CODE' =
================================================
0     ATL
1     ORD
...
49    IND
Name: Code
```

```
================================================
= SELECTED COLUMNS WITH FILTERED ROWS =
================================================
   Code  2001 Total                                          Airport
0   ATL    36384264  Atlanta, GA (Hartsfield-Jackson Atlanta Intern...
1   ORD    28626694          Chicago, IL (Chicago O'Hare International)
...
18  BOS    10016801  Boston, MA (General Edward Lawrence Logan Inte...
```

```
================================================
= COLUMN TOTALS =
================================================
2001 Total    492424399
2010 Total    558240706

================================================
= 'CODE' COLUMN SET AS INDEX =
================================================
                                          Airport Code  2001 Rank  2001 Total  2010 Rank
2010 Total  2011 Rank     Total  Percent change 2001-2011  Percent change 2010-2011
0   Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ATL          1    36384264          1
42655392          1  33034788                      -9.2                    -22.6
1          Chicago, IL (Chicago O'Hare International)  ORD          2    28626694          2
30033313          2  22367052                     -21.9                    -25.5
2        Dallas, TX (Dallas/Fort Worth International)  DFW          3    25198256          3
26785739          3  20430281                     -18.9                    -23.7
3                  Denver, CO (Denver International)  DEN          6    16397390          4
24965553          4  19190341                      17.0                    -23.1
4         Los Angeles, CA (Los Angeles International)  LAX          4    22873307          5
22860849          5  18379418                     -19.6                    -19.6
...
49   2817684                     12.5                    -22.0
49      Indianapolis, IN (Indianapolis International)  IND         48     3410636         50
3716884         50   2750105                     -19.4               -26.0
```

Chapter 19 Exercises

Exercise 19-1

Create a DataFrame with columns named 'Test1', 'Test2', up to 'Test6'. Use default row indexes. Fill the DataFrame with random values.

a. Print only columns 'Test3' and 'Test5'.

b. Print the dataframe with every value multiplied by 3.6

Exercise 19-2

The file parasite_data.csv, in the DATA folder, has some results from analysis on some intestinal parasites (not that it matters for this exercise...).

Read parasite_data.csv into a DataFrame. Print out all rows where the Shannon Diversity is >= 1.0.

Chapter 20

matplotlib

Chapter 20 Objectives

- **Understand what matplotlib can do**
- **Create many kinds of plots**
- **Label axes, plots, and design callouts**

About matplotlib

- **matplotlib is a package for making 2D plots**
- **Emulates MATLAB®, but not a drop-in replacement**
- **matplotlib's philosophy: create simple plots simply**
- **Plots are publication quality**
- **Plots can be rendered in GUI applications**

This chapter's discussion of **matplotlib** will use the iPython notebook named MatplotlibExamples.ipynb. Please start the iPython notebook server and load this notebook, as directed by the instructor.

matplotlib architecture

- **pylab/pyplot** ***front end plotting functions***
- **API** ***create/manage figures, text, plots***
- **backends** ***device-independent renderers***

matplotlib consists of roughly three parts: pylab/pyplot, the API, and and the backends.

pyplot is a set of functions which allow the user to quickly create plots. Pyplot functions are named after similar functions in MATLAB.

pylab combines pyplot with **numpy**. This makes pylab emulate MATLAB more closely, and thus is good for interactive use, e.g., with **iPython**. On the other hand, **pyplot** alone is very convenient for scripting. The main advantage of pylab is that it imports methods from both **pyplot** and **pylab**.

The API is a large set of classes that do all the work of creating and manipulating plots, lines, text, figures, and other graphic elements. The API can be called directly for more complex requirements.

There are many backends which render the in-memory representation, created by the API, to a video display or hardcopy format. For example, backends include PS for PostScript, SVG for scalable vector graphics, and PDF.

The **--pylab** option to iPython imports matplotlib.pyplot as plt, and numpy as np

matplotlib Terminology

- **Figure**
- **Axis**
- **Subplot**

A Figure is one "picture". It has a border ("frame"), and other attributes. A Figure can be saved to a file.

A Plot is one set of values graphed onto the Figure. A Figure can contain more than one Plot.

Axes and Subplot are similar; the difference is how they get placed on the figure. Subplots allow multiple plots to be placed in a rectangular grid. Axes allow multiple plots to placed at any location, including within other plots, or overlapping.

matplotlib uses default objects for all of these, which are sufficient for simple plots. You can explicitly create any or all of these objects to fine-tune a graph. Most of the time, for simple plots, you can accept the defaults and get great-looking figures.

matplotlib Keeps State

- **Primary method is matplotlib.pyplot()**
- **The current figure can have more than one plot**
- **Calling show() displays the current figure**

matplotlib.pyplot is the workhorse of figure drawing. It is usually aliased to "pl" or "plt".

Calling plt.plot() plots one set of data on the current figure. Calling it again adds another plot to the same figure.

Plt.show() displays the figure, although iPython may display each separate plot, depending on the current settings.

You can pass one or two datasets to plot(). If there are two datasets, they need to be the same length, and represent the x and y data.

What Else Can You Do?

- **Multiple plots**
- **Control ticks on any axis**
- **Scatter plots**
- **Polar axes**
- **3D Plots**
- **Quiver plots**
- **Pie Charts**

There are many other types of drawings that matplotlib can create. Also, there are many more style details that can be tweaked.

See http://matplotlib.org/gallery.html for dozens of sample plots and their source.

Chapter 20 Labs

Exercise 20-1

Using the file energy_use_quad.csv in the DATA folder, use matplotlib to plot the data for "Transportation", "Industrial", and "Residential and Commercial". Don't plot the "as a percent...".

You can do this in iPython, or as a standalone script. If you create a standalone script, save the figure to a file, so you can view it.

Use **pandas** to read the data. The columns are, in Python terms:

```
['Desc',"1960","1965","1970","1975","1980","1985","1990",
"1991","1992","1993","1994","1995","1996","1997","1998","
1999","2000","2001","2002","2003","2004","2005","2006","2
007","2008","2009","(R) 2010","2011"]
```

Hint: See the script pandas_energy.py in the EXAMPLES folder to see how to load the data.

Chapter 21
The Python Imaging Library

Chapter 21 Objectives

- **Get an overview of the Python Imaging Library**
- **Open and save image files in various formats**
- **Create image thumbnails**
- **Crop, cut, and paste images**
- **Transform images with filters**

The Python Imaging Library

- **Adds imaging processing to the Python interpreter**
- **Possible uses**
  - **Image Archives (thumbnails, resizing, etc.)**
  - **Image Display (interfaces with GUIs)**
  - **Image Processing (filtering, conversion, rotating, etc.)**

The Python Imaging library (PIL) is a comprehensive package for working with images. It has support for a large number of image types.

Supported Image File Types

BMP	MPEG (identify only)
BUFR (identify only)	MSP
CUR (read only)	PALM (write only)
DCX (read only)	PCD (read only)
EPS (write-only)	PCX
FITS (identify only)	PDF (write only)
FLI, FLC (read only)	PIXAR (read only)
FPX (read only)	PNG
GBR (read only)	PPM
GD (read only)	PSD (read only)
GIF	SGI (read only)
GRIB (identify only)	SPIDER
HDF5 (identify only)	TGA (read only)
ICO (read only)	TIFF
IM	WAL (read only)
IMT (read only)	WMF (identify only)
IPTC/NAA (read only)	XBM
JPEG	XPM (read only)
MCIDAS (read only)	XV Thumbnails
MIC (read only)	

The Image class

- **Primary class of PIL**
- **Used to process and manipulate the image**
- **Use attributes to examine image**
- **Can load from/save to file**

The Image class is the main object of the PIL. An image is typically loaded from a file into an Image object, and from there it can be processed.

An image has several attributes that provide detailed information – format, size, mode, etc.

Once the image is loaded, there are many methods for working with the image.

Reading and writing

- **Open with Image.open(*filename*)**
- **Save with Image.save(*filename*, *[filetype])***
- **Filename determines saved format**

To get started, open an image with Image.open(). This returns an Image object, which is the basis for all processing.

The image can then be manipulated, and eventually saved in the same, or a different, format.

Open() only reads the file header, so opening a file is very fast. PIL only reads the actual content of the file if necessary.

Example

pil_basics.py

```
import Image

im = Image.open('../DATA/felix_auto.jpeg')
print im.format
print im.size
print im.mode

im.save('felix_auto.png')
```

```
pil_basics.py
JPEG
(3008, 2000)
RGB
```

You can also use a file-like object in place of a filename

Creating thumbnails

- **Use *IM*.thumbnail(*size*)**
- **Size is tuple of height, width**

To convert an image into a thumbnail, call the thumbnail() method. Pass in a tuple of height and width, then save in desired format.

Example

pil_thumb.py

```
import Image

size = 125,125
im = Image.open('../DATA/felix_auto.jpeg')
im.thumbnail(size)
im.save('felix_auto_small.jpg')
```

Coordinate System

- **Points: (0,0) is upper left**
- **Rectangles: (0, 0, 800, 800) is entire 800x600 pixel image**

The Python Imaging Library uses a Cartesian pixel coordinate system, with (0,0) in the upper left corner. Note that the coordinates refer to the implied pixel corners; the center of a pixel addressed as (0, 0) actually lies at (0.5, 0.5).

Coordinates are usually passed to the library as 2-tuples (x, y). Rectangles are represented as 4-tuples, with the upper left corner given first. For example, a rectangle covering all of an 800x600 pixel image is written as (0, 0, 800, 600).

Cropping and pasting

- **Use crop() to pull out a region from an image**
- **Use paste() to insert an image into another**

The Image class contains methods allowing you to manipulate regions within an image. To extract a sub-rectangle from an image, use the crop() method. crop() requires a tuple containing the bounds of the region.

Use paste() to insert a previously extracted region.

Example

pil_crop.py

```
import Image

box = (1200, 100, 2000, 400)
im = Image.open('../DATA/felix_auto.jpeg')
region = im.crop(box)
region = region.transpose(Image.FLIP_LEFT_RIGHT)
im.paste(region,box)
im.save('felix_auto_cropped.jpg')
```

felix_auto.jpeg

felix_auto_cropped.jpg

Rotating, resizing, and flipping

- **Use IM.resize(), IM.rotate(), or im.transpose()**
- **These return new Image objects**

The Image class contains several methods for transposing and resizing images.

.resize() takes a tuple giving the new size. .rotate() takes the angle in degrees to rotate counter-clockwise. .transpose() is a convenience method for flipping or rotating an image.

To rotate the image in 90 degree steps, you can use transpose(), which can also be used to flip an image vertically or horizontally.

Example

pil_transpose.py

```
import Image
SMALLSIZE = 200,200
im = Image.open('../DATA/felix_auto.jpeg')
im_small = im.resize(SMALLSIZE)

im1 = im_small.rotate(45)
im1.save('felix_auto_45.jpg')

im2 = im.transpose(Image.FLIP_LEFT_RIGHT)
im2.save('felix_auto_flipLR.jpg')

im3 = im.transpose(Image.FLIP_TOP_BOTTOM)
im3.save('felix_auto_flipTB.jpg')
```

felix_auto_45.jpg

felix_auto_flipLR.jpg

felix_auto_flipTB.jpg

Enhancing

- **Many filters provided**
- **ImageFilter**
  - **filter(), point()**
- **ImageEnhance**
  - **contrast(), etc.**
- **New Image object is returned**

PIL has many filters to change the appearance of an image. These are in the ImageFilter module.

The point method can be used to translate the pixel values of an image (e.g. image contrast manipulation). Pass a function expecting one argument to this method. Each pixel is processed according to that function

You can quickly apply any simple conversion to an image, or combine point and paste to modify a region within an image.

The ImageEnhance module has functions to further process an image.

Examples

pyfilter.py

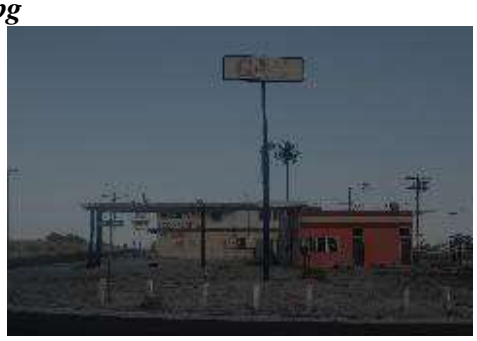
```
import Image, ImageFilter, ImageEnhance

SMALLSIZE = 300,200
im = Image.open('../DATA/felix_auto.jpeg')
im_small = im.resize(SMALLSIZE)

im_emboss = im_small.filter(ImageFilter.EMBOSS)
im_emboss.save('felix_auto_emboss.jpg')

im_blur = im_small.filter(ImageFilter.BLUR)
im_blur.save('felix_auto_blur.jpg')

enh_bright = ImageEnhance.Brightness(im_small)
im_bright = enh_bright.enhance(2.5)
im_bright.save('felix_auto_bright.jpg')

enh_contrasty = ImageEnhance.Contrast(im_small)
im_contrasty = enh_bright.enhance(.5)
im_contrasty.save('felix_auto_contrasty.jpg')
```

felix_auto_emboss.jpg

felix_auto_blur.jpg

felix_auto_bright.jpg

felix_auto_lowcontrast.jpg



Chapter 21 Exercises

Exercise 21-1

Open the image salad.jpeg in the DATA folder. Save it as exactly half its original size.

Open salad.jpeg, flip it horizontally, and save.

Open salad.jpeg, make a 50x50 thumbnail, and save.

(In all cases, save with a new name – don't overwrite existing files).

Appendix A
Python Bibliography

Python Bibliography

Title	Author	Publisher
Python Essential Reference, 4th. Ed.	David M. Beazley	Addison-Wesley Professional
The Python Standard Library By Example	Doug Hellmann	Addison-Wesley Professional
Learning Python, 2nd Ed.	Mark Lutz, David Asher	O'Reilly & Assoc.
Python Programming on Win32	Mark Hammond, Andy Robinson	O'Reilly & Assoc.
Python Cookbook, 2nd. Ed.	Alex Martelli, Anna Ravenscroft, David Ascher	O'Reilly & Assoc.
Python in a Nutshell	Alex Martelli	O'Reilly & Assoc.
Python: How to Program	Harvey M. Deitel, Paul J. Deitel, Jonathan P. Liperi, Ben Wiedermann	Prentice Hall
Programming Python, 2nd Ed.	Mark Lutz	O'Reilly & Assoc.

Appendix B

Python Gotchas

"Too little freedom makes life confusingly clumsy; too much, clumsily confusing. Luckily, the tension between freedom and restraint eventually gets severed by Guido's Razor."

--Tim Peters

Python has a very clean syntax, and few rough edges. However, there are a few 'gotchas' that can trip up a new programmer:

Gotcha #1: Backslashes are escape characters

This occurs when Windows programmers specify file names in strings. "\\spam\eggs" comes out as "\spam\eggs". Raw strings work OK (r"\\spam\eggs").

If the path ends with a backslash, however, you're pretty much doomed; even in a raw string, a backslash prevents the quote character (single or double) currently in use from being interpreted. So a raw string cannot end with a backslash.

Gotcha #2: The quotient of integers is an integer

The gotcha is this — if you divide an integer by an integer, you get an integer. If the division produces a remainder, the result is truncated downward. (Note that this is true truncation, not rounding toward zero.)

```
print 5/3       produces 1
print (-5)/3    produces -2
print 5/3.0     produces 1.66666666667, because 3.0 is a float, not an integer
```

Gotcha #3: A comma at the end of a print statement appears to write a trailing space.

It really causes an immediately subsequent *print* statement to write a leading space!

The Python Language Reference Manual says, about the *print* statement,

> **A "\n" character is written at the end, unless the print statement ends with a comma.**

But it also says that if two **print** statements in succession write to **stdout**, and the first one ends with a comma (and so doesn't write a trailing newline), then the second one prepends a leading space to its output.

Gotcha #4: Be careful of the argument order when using **sub()** or **subn()** from **re**

When using the sub(), or subn() methods from an re object, the replacement text is the *first* argument, followed by the string in which to perform the replacement:

```
s = "# this is a comment"
foo = re.compile("^#")
bar = foo.sub("//",s)    # not bar = foo.sub(s,"//")
```

Gotcha #5: Don't overwrite builtins

Python does not prevent you from overwriting builtin functions like this:

```
import sys

dir = sys.argv[1]
```

Use **pylint** or a similar tool to warn you when this happens.

Appendix C

Builtin functions

Builtin Functions

abs(*x*)

Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

apply(*function, args*[*, keywords*])

argument list; the number of arguments is the the length of the tuple. If the optional *keywords* argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to be added to the end of the the argument list. Calling `apply()` is different from just calling *function*`(`*args*`)`, since in that case there is always exactly one argument. The use of `apply()` is equivalent to *function*`(*`*args*`, **`*keywords*`)`.

buffer(object[, offset[, size]])

The *object* argument must be an object that supports the buffer call interface (such as strings, arrays, and buffers). A new buffer object will be created which references the *object* argument. The buffer object will be a slice from the beginning of *object* (or from the specified *offset*). The slice will extend to the end of *object* (or will have a length given by the *size* argument).

callable(object)

Return true if the *object* argument appears callable, false if not. If this returns true, it is still possible that a call fails, but if it is false, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); class instances are callable if they have a `__call__()` method.

chr(i)

Return a string of one character whose ASCII code is the integer *i*. For example, `chr(97)` returns the string `'a'`. This is the inverse of `ord()`. The argument must be in the range [0..255], inclusive; `ValueError` will be raised if *i* is outside that range.

cmp(x, y)

Compare the two objects *x* and *y* and return an integer according to the outcome. The return value is negative if *x* < *y*, zero if *x* == *y* and strictly positive if *x* > *y*.

coerce(x, y)

Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations.

compile(string, filename, kind[, flags[, dont_inherit]])

Compile the *string* into a code object. Code objects can be executed by an `exec` statement or evaluated by a call to `eval()`. The *filename* argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file (`'<string>'` is commonly used). The *kind* argument specifies what kind of code must be compiled; it can be `'exec'` if *string* consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something else than `None` will printed).

When compiling multi-line statements, two caveats apply: line endings must be represented by a single newline character (`'\n'`), and the input must be terminated by at least one newline character. If line endings are represented by `'\r\n'`, use the string `replace()` method to change them into `'\n'`.

The optional arguments *flags* and *dont_inherit* (which are new in Python 2.2) control which future statements (see PEP 236) affect the compilation of *string*. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling compile. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it - the future statements in effect around the call to compile are ignored.

Future statemants are specified by bits which can be bitwise or-ed together to specify multiple statements. The bitfield required to specify a given feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the `__future__` module.

complex(real[, imag])

Create a complex number with the value *real* + *imag**j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If *imag* is omitted, it defaults to zero and the function serves as a numeric conversion function like `int()`, `long()` and `float()`.

delattr(object, name)

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(`*x*`, '`*foobar*`')` is equivalent to `del` *x*`.`*foobar*.

dict([mapping-or-sequence])

Return a new dictionary initialized from the optional argument. If an argument is not specified, return a new empty dictionary. If the argument is a mapping object, return a dictionary mapping the same keys to the same values as does the mapping object. Else the argument must be a sequence, a container that supports iteration, or an iterator object. The elements of the argument must each also be of one of those kinds, and each must in turn contain exactly two objects. The first is used as a key in the new dictionary, and the second as the key's value. If a given key is seen more than once, the last value associated with it is retained in the new dictionary. For example, these all return a dictionary equal to `{1: 2, 2: 3}`:

- `dict({1: 2, 2: 3})`
- `dict({1: 2, 2: 3}.items())`
- `dict({1: 2, 2: 3}.iteritems())`
- `dict(zip((1, 2), (2, 3)))`
- `dict([[2, 3], [1, 2]])`
- `dict([(i-1, i) for i in (2, 3)])`

New in version 2.2.

dir([object])

Without arguments, return the list of names in the current local symbol table. With an argument, attempts to return a list of valid attributes for that object. This information is gleaned from the object's `__dict__` attribute, if defined, and from the class or type object. The list is not necessarily complete. If the object is a module object, the list contains the names of the module's attributes. If

the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases. Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes. The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)
['__doc__', '__name__', 'calcsize', 'error', 'pack', 'unpack']
```

Note: Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases.

`divmod(a, b)`

Take two numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as `(a / b, a % b)`. For floating point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to *a*, if `a % b` is non-zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

`eval(expression[, globals[, locals]])`

The arguments are a string and two optional dictionaries. The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local name space. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where `eval` is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case pass a code object instead of a string. The code object must have been compiled passing `'eval'` as the *kind* argument.

Hints: dynamic execution of statements is supported by the `exec` statement. Execution of statements from a file is supported by the `execfile()` function. The `globals()` and `locals()` functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `execfile()`.

execfile(file[, globals[, locals]])

This function is similar to the `exec` statement, but parses a file instead of a string. It is different from the `import` statement in that it does not use the module administration -- it reads the file unconditionally and does not create a new module.[2.2]

The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the *globals* and *locals* dictionaries as global and local namespace. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where `execfile()` is called. The return value is `None`.

Warning: The default *locals* act as described for function `locals()` below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `execfile()` returns. `execfile()` cannot be used reliably to modify a function's locals.

file(filename[, mode[, bufsize]])

Return a new file object (described earlier under Built-in Types). The first two arguments are the same as for `stdio`'s `fopen()`: *filename* is the file name to be opened, *mode* indicates how the file is to be opened: `'r'` for reading, `'w'` for writing (truncating an existing file), and `'a'` opens it for appending (which on *some* Unix systems means that *all* writes append to the end of the file, regardless of the current seek position).

Modes `'r+'`, `'w+'` and `'a+'` open the file for updating (note that `'w+'` truncates the file). Append `'b'` to the mode to open the file in binary mode, on systems that differentiate between binary and text files (else it is ignored). If the file cannot be opened, `IOError` is raised.

If *mode* is omitted, it defaults to `'r'`. When opening a binary file, you should append `'b'` to the *mode* value for improved portability. (It's useful even on systems which don't treat binary and text files differently, where it serves as documentation.) The optional *bufsize* argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative *bufsize* means to use the system default, which is usually line buffered for for tty devices and fully buffered for other files. If omitted, the system default is used.[2.3]

The `file()` constructor is new in Python 2.2. The previous spelling, `open()`, is retained for compatibility, and is an alias for `file()`.

filter(function, list)

Construct a list from those elements of *list* for which *function* returns true. *list* may be either a sequence, a container which supports iteration, or an iterator, If *list* is a string or a tuple, the result also has that type; otherwise it is always a list. If *function* is `None`, the identity function is assumed, that is, all elements of *list* that are false (zero or empty) are removed.

float(x)

Convert a string or a number to floating point. If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace; this behaves identical to `string.atof(`*x*`)`. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned.

Note: When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. The specific set of strings accepted which cause these values to be returned depends entirely on the C library and is known to vary.

getattr(object, name[, default])

Return the value of the named attributed of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise `AttributeError` is raised.

globals()

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

hasattr(object, name)

The arguments are an object and a string. The result is 1 if the string is the name of one of the object's attributes, 0 if not. (This is implemented by calling `getattr(`*object*`, `*name*`)` and seeing whether it raises an exception or not.)

hash(object)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

help([object])

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

hex(x)

Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression. Note: this always yields an unsigned literal. For example, on a 32-bit machine, `hex(-1)` yields `'0xffffffff'`. When evaluated on a machine with the same word size, this literal is evaluated as -1; at a different word size, it may turn up as a large positive number or raise an `OverflowError` exception.

id(object)

Return the `identity' of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects whose lifetimes are disjunct may have the same `id()` value. (Implementation note: this is the address of the object.)

input([prompt])

Equivalent to `eval(raw_input(`*prompt*`))`. **Warning:** This function is not safe from user errors! It expects a valid Python expression as input; if the input is not syntactically valid, a `SyntaxError` will be raised. Other exceptions may be raised if there is an error during evaluation. (On the other hand, sometimes this is exactly what you need when writing a quick script for expert use.)

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Consider using the `raw_input()` function for general input from users.

int(x[, radix])

Convert a string or number to a plain integer. If the argument is a string, it must contain a possibly signed decimal number representable as a Python integer, possibly embedded in whitespace; this behaves identical to `string.atoi(x[, radix])`. The *radix* parameter gives the base for the conversion and may be any integer in the range [2, 36], or zero. If *radix* is zero, the proper radix is guessed based on the contents of string; the interpretation is the same as for integer literals. If *radix* is specified and *x* is not a string, `TypeError` is raised. Otherwise, the argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers truncates (towards zero).

intern(string)

Enter *string* in the table of ``interned'' strings and return the interned string - which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup - if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys. Interned strings are immortal (never get garbage collected).

isinstance(object, classinfo)

Return true if the *object* argument is an instance of the *classinfo* argument, or of a (direct or indirect) subclass thereof. Also return true if *classinfo* is a type object and *object* is an object of that type. If *object* is not a class instance or a object of the given type, the function always returns false. If *classinfo* is neither a class object nor a type object, it may be a tuple of class or type objects, or may recursively contain other such tuples (other sequence types are not accepted). If *classinfo* is not a class, type, or tuple of classes, types, and such tuples, a `TypeError` exception is raised. Changed in version 2.2: Support for a tuple of type information was added.

issubclass(class1, class2)

Return true if *class1* is a subclass (direct or indirect) of *class2*. A class is considered a subclass of itself. If either argument is not a class object, a `TypeError` exception is raised.

iter(o[, sentinel])

Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *o* must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at `0`). If it does not support either of those protocols, `TypeError` is raised. If the second argument, *sentinel*, is given, then *o* must be a callable object. The iterator created in this case will call *o* with no arguments for each call to its `next()` method; if the value returned is equal to *sentinel*, `StopIteration` will be raised, otherwise the value will be returned. New in version 2.2.

len(s)

Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

list([sequence])

Return a list whose items are the same and in the same order as *sequence*'s items. *sequence* may be either a sequence, a container that supports iteration, or an iterator object. If *sequence* is already a list, a copy is made and returned, similar to *sequence*`[:]`. For instance, `list('abc')` returns `['a', 'b', 'c']` and `list( (1, 2, 3) )` returns `[1, 2, 3]`.

locals()

Return a dictionary representing the current local symbol table. **Warning:** The contents of this dictionary should not be modified; changes may not affect the values of local variables used by the interpreter.

long(x[, radix])

Convert a string or number to a long integer. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace; this behaves identical to `string.atol(x)`. The *radix* argument is interpreted in the same way as for `int()`, and may only be given when *x* is a string. Otherwise, the argument may be a plain or long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point

numbers to integers truncates (towards zero).

map(function, list, ...)

Apply *function* to every item of *list* and return a list of the results. If additional *list* arguments are passed, *function* must take that many arguments and is applied to the items of all lists in parallel; if a list is shorter than another it is assumed to be extended with `None` items. If *function* is `None`, the identity function is assumed; if there are multiple list arguments, `map()` returns a list consisting of tuples containing the corresponding items from all lists (a kind of transpose operation). The *list* arguments may be any kind of sequence; the result is always a list.

max(s[, args...])

With a single argument *s*, return the largest item of a non-empty sequence (such as a string, tuple or list). With more than one argument, return the largest of the arguments.

min(s[, args...])

With a single argument *s*, return the smallest item of a non-empty sequence (such as a string, tuple or list). With more than one argument, return the smallest of the arguments.

oct(x)

Convert an integer number (of any size) to an octal string. The result is a valid Python expression. Note: this always yields an unsigned literal. For example, on a 32-bit machine, `oct(-1)` yields `'037777777777'`. When evaluated on a machine with the same word size, this literal is evaluated as -1; at a different word size, it may turn up as a large positive number or raise an `OverflowError` exception.

open(filename[, mode[, bufsize]])

An alias for the `file()` function above.

ord(c)

Return the ASCII value of a string of one character or a Unicode character. E.g., `ord('a')` returns the integer `97`, `ord(u'
u2020')` returns `8224`. This is the inverse of `chr()` for strings and of `unichr()` for Unicode characters.

pow(x, y[, z])

Return *x* to the power *y*; if *z* is present, return *x* to the power *y*, modulo *z* (computed more efficiently than `pow(x, y) % z`). The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised.) If the second argument is negative, the third argument must be omitted. If *z* is present, *x* and *y* must be of integer types, and *y* must be non-negative. (This restriction was added in Python 2.2. In Python 2.1 and before, floating 3-argument `pow()` returned platform-dependent results depending on floating-point rounding accidents.)

range([start,] stop[, step])

This is a versatile function to create lists containing arithmetic progressions. It is most often used in `for` loops. The arguments must be plain integers. If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. The full form returns a list of plain integers `[`*start*`,` *start* `+` *step*`,` *start* `+ 2 *` *step*`, ...]`. If *step* is positive, the last element is the largest *start* `+` *i* `*` *step* less than *stop*; if *step* is negative, the last element is the largest *start* `+` *i* `*` *step* greater than *stop*. *step* must not be zero (or else `ValueError` is raised). Example:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

raw_input([prompt])

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `raw_input()` will use it to provide elaborate line editing and history features.

reduce(function, sequence[, initializer])

Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `(((1+2)+3)+4)+5)`. If the optional *initializer* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

reload(module)

Re-parse and re-initialize an already imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

There are a number of caveats:

If a module is syntactically correct but its initialization fails, the first `import` statement for it does not bind its name locally, but does store a (partially initialized) module object in `sys.modules`. To reload the module you must first `import` it again (this will bind the name to the partially initialized module object) before you can `reload()` it.

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects -- with a `try` statement it can test for the table's presence and skip its initialization if desired.

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for `sys`, `__main__` and `__builtin__`. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` ... `import` ..., calling `reload()` for the other module does not redefine the objects imported from it -- one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module.name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances -- they continue to use the old class definition. The same is true for derived classes.

repr(object)

Return a string containing a printable representation of an object. This is the same value yielded by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`.

round(x[, n])

Return the floating point value *x* rounded to *n* digits after the decimal point. If *n* is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus *n*; if two multiples are equally close, rounding is done away from 0 (so. for example, `round(0.5)` is `1.0` and `round(-0.5)` is `-1.0`).

setattr(object, name, value)

This is the counterpart of `getattr()`. The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(`*x*`, '`*foobar*`', 123)` is equivalent to *x*`.`*foobar* `= 123`.

slice([start,] stop[, step])

Return a slice object representing the set of indices specified by `range(`*start*`, `*stop*`, `*step*`)`. The *start* and *step* arguments default to None. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `"a[start:stop:step]"` or `"a[start:stop, i]"`.

str(object)

Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with `repr(object)` is that `str(object)` does not always attempt to return a string that is acceptable to `eval()`; its goal is to return a printable string.

tuple([sequence])

Return a tuple whose items are the same and in the same order as *sequence*'s items. *sequence* may be a sequence, a container that supports iteration, or an iterator object. If *sequence* is already a tuple, it is returned unchanged. For instance, `tuple('abc')` returns returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`.

type(object)

Return the type of an *object*. The return value is a type object. The standard module `types` defines names for all built-in types. For instance:

```
>>> import types
>>> if type(x) == types.StringType: print "It's a string"
```

unichr(i)

Return the Unicode string of one character whose Unicode code is the integer *i*. For example, `unichr(97)` returns the string `u'a'`. This is the inverse of `ord()` for Unicode strings. The argument must be in the range [0..65535], inclusive. `ValueError` is raised otherwise. New in version 2.0.

unicode(object[, encoding[, errors]])

Return the Unicode string version of *object* using one of the following modes:

If *encoding* and/or *errors* are given, `unicode()` will decode the object which can either be an 8-bit string or a character buffer using the codec for *encoding*. The *encoding* parameter is a string giving the name of an encoding. Error handling is done according to *errors*; this specifies the treatment of characters which are invalid in the input encoding. If *errors* is `'strict'` (the default), a `ValueError` is raised on errors, while a value of `'ignore'` causes errors to be silently ignored, and a value of `'replace'` causes the official Unicode replacement character, `U+FFFD`, to be used to replace input characters which cannot be decoded. See also the `codecs` module.

If no optional parameters are given, `unicode()` will mimic the behaviour of `str()` except that it returns Unicode strings instead of 8-bit strings. More precisely, if *object* is an Unicode string or subclass it will return a Unicode string without any additional decoding applied. For objects which provide a `__unicode__` method, it will call this method without arguments to create a Unicode string. For all other objects, the 8-bit string version or representation is requested and then converted to a Unicode string using the codec for the default encoding in `'strict'` mode. New in version 2.0.

vars([object])

Without arguments, return a dictionary corresponding to the current local symbol table. With a module, class or class instance object as argument (or anything else that has a `__dict__` attribute), returns a dictionary corresponding to the object's symbol table. The returned dictionary should not be modified: the effects on the corresponding symbol table are undefined.[2.4]

`xrange([start,] stop[, step])`

This function is very similar to `range()`, but returns an ``xrange object'' instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of `xrange()` over `range()` is minimal (since `xrange()` still has to create the values when asked for them) except when a very large range is used on a memory-starved machine or when all of the range's elements are never used (such as when the loop is usually terminated with `break`).

`zip(seq1, ...)`

This function returns a list of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences. At least one sequence is required, otherwise a `TypeError` is raised. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple argument sequences which are all of the same length, `zip()` is similar to `map()` with an initial argument of `None`. With a single sequence argument, it returns a list of 1-tuples. New in version 2.0.

Appendix D

Setting Up Komodo Edit

About Komodo Edit

- **Free editor**
- **Context-sensitive**
- **Syntax highlighting**
- **Edit-Run-Debug automation**
- **Projects**
- **Macros**
- **Cross-platform**

Komodo Edit is a free IDE (integrated development environment) from ActiveState Software. It is the community version of their Komodo IDE, which has some value-added features.

It includes Code Intelligence, which provides autocompletion, calltips, and the Code Browser.

It will syntax highlight your Python source code, and provides real-time syntax checking. It automates the edit-run-debug process via customizable macros. The macro facility allows you to automate nearly any task, as well.

You can create projects that contain multiple files and folders. This is very convenient for large applications.

Komodo Edit runs on Linux, Windows, and MacOS. The current version is 6.1.3.

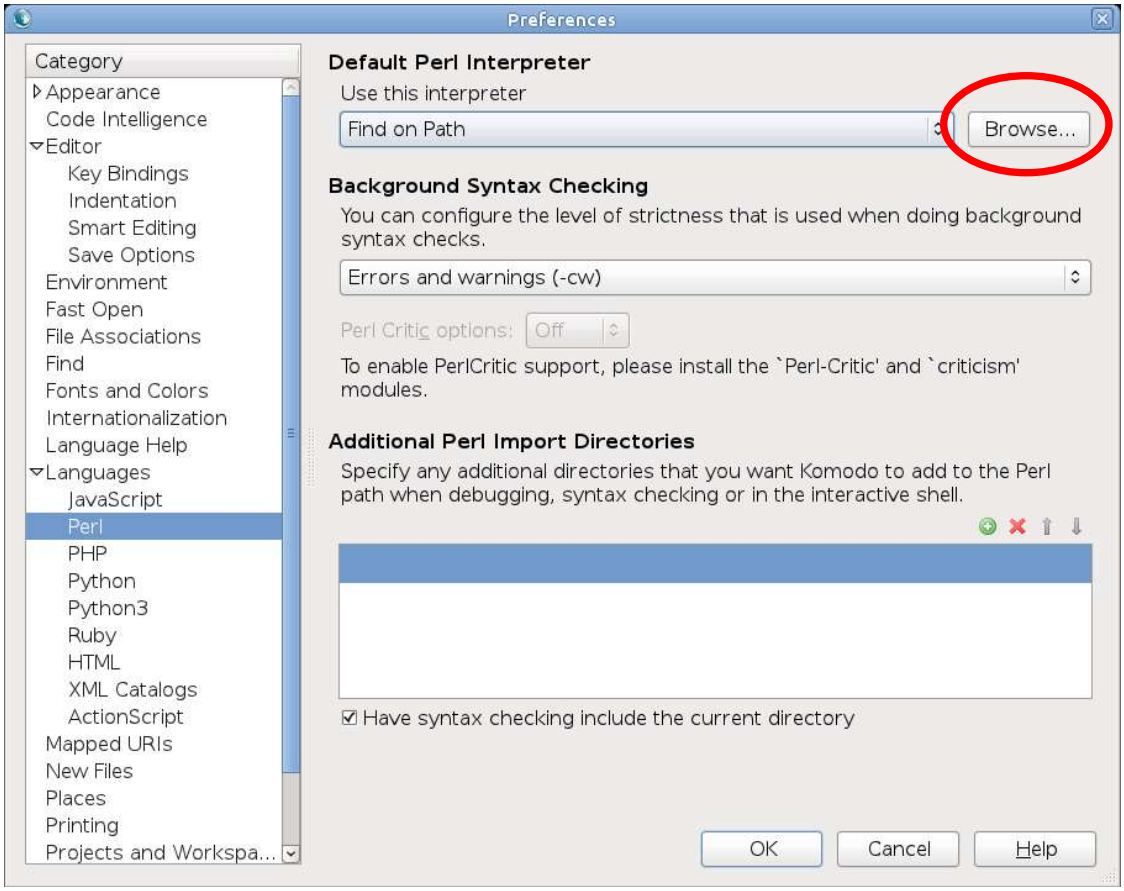
Installing Komodo Edit

To install Komodo Edit, go to www.activestate.com/komodo-edit/downloads and download the appropriate Komodo Edit 6 installer for your platform. The installer will do all the work. You should be root on Linux, or administrator on Windows, although this is not absolutely required.

Setting up Komodo Edit for Python

While it will look for your Python executable using the PATH variable, it is best to tell Komodo exactly which version of Python you're using. This is especially important if you have more than one version of Python installed.

From the **Edit** menu, select **Preferences** to display the **Preferences** dialog. Click on **Languages** in the **Category** list. Select **Python** from the list of languages. It should look like this:



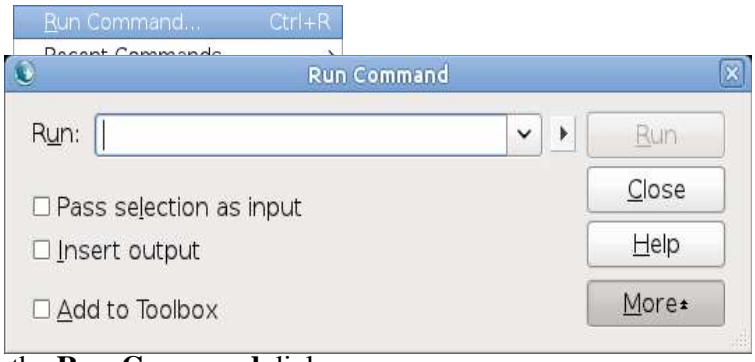
Click **Browse**, then find the Python executable that you want Komodo to use. On Linux, the default executable is **/usr/bin/python**. On Windows, the default is **C:\Python\bin\Python.exe**

Click **OK** to close the dialog.

Creating a Run command

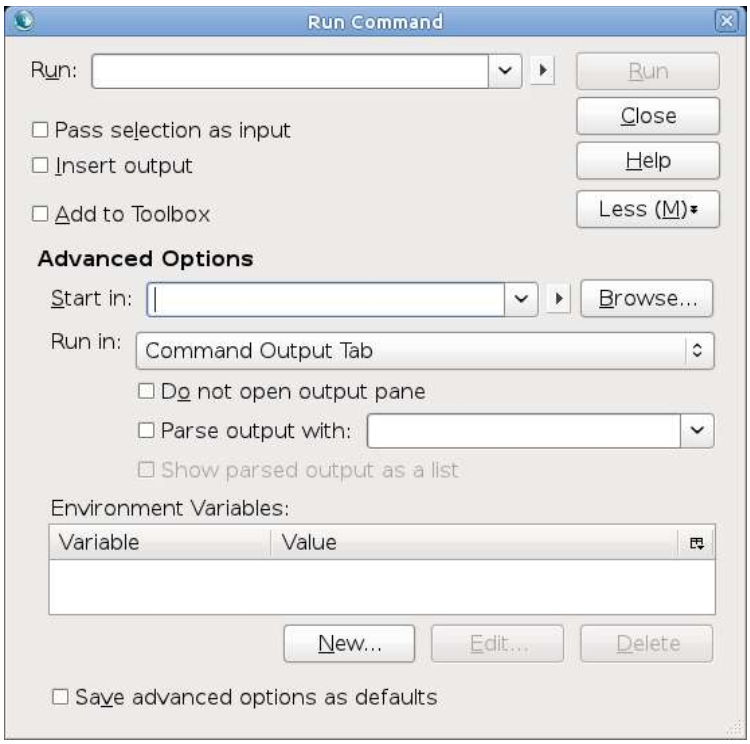
To make it easy to run scripts during development, you'll want to set up one or more commands. This will allow you to run your script "inside" Komodo. The output will display in Komodo's **Command Output** tab, or in a new console window (OS prompt).

To get started, load any Python script. Select **Run Command...** from the **Tools** menu (or just type Control-R).



You'll get the **Run Command** dialog.

Click on **More**, to expand the dialog so it looks like this:



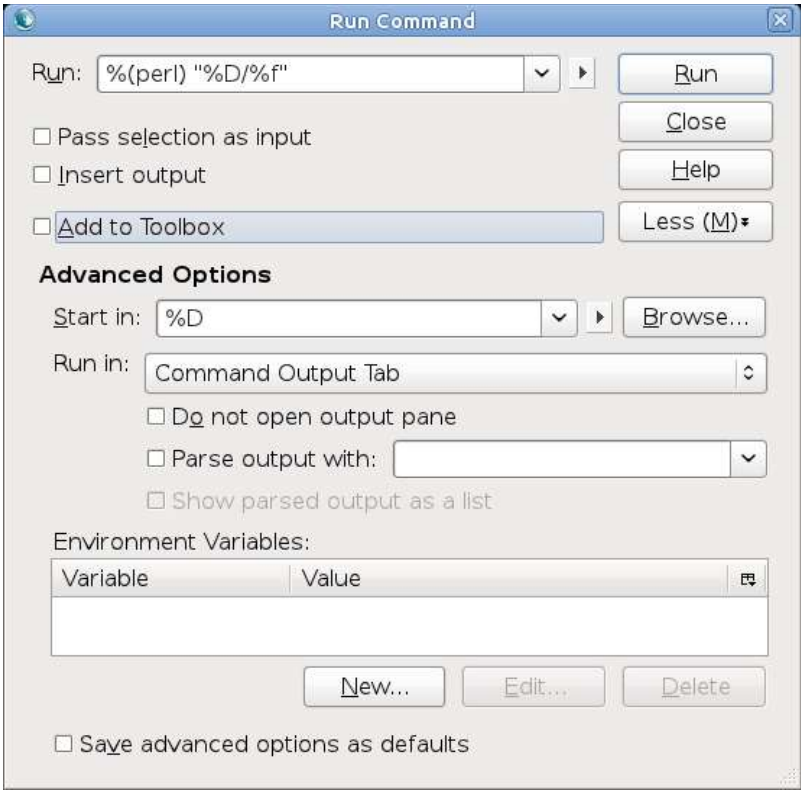
1. In the **Run:** entry, add the following text:

```
%(python) -u "%D/%f"
```

2. Check **Add to Toolbox**
3. In the Start in: entry, add the following text:

```
%D
```

4. The dialog should look like this:



5. Click Run. This will both run your script and create a macro.

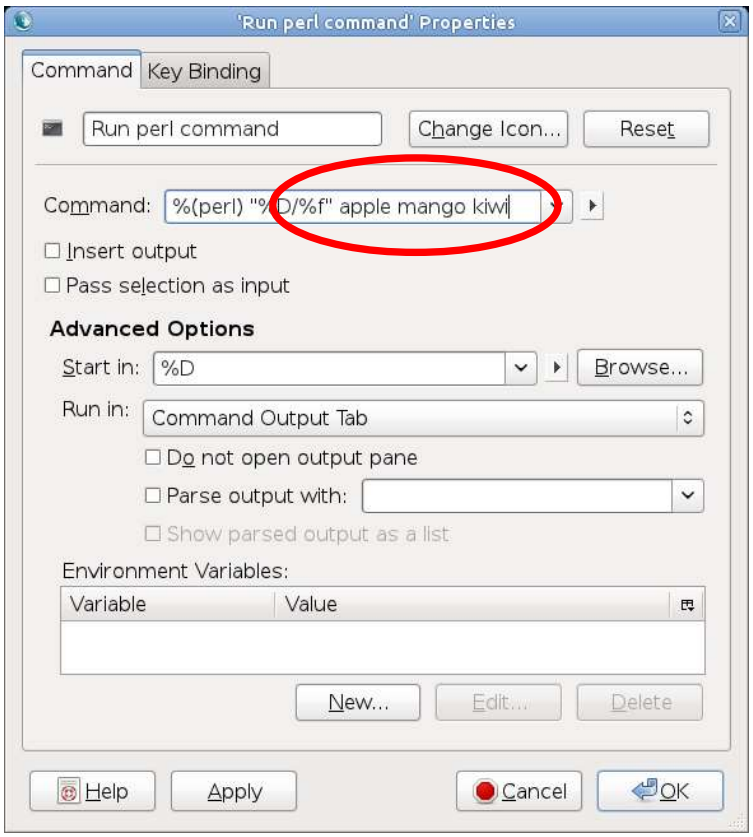
Notice that the **Toolbox** window appears on the right (you can show/hide this from the **View/Tabs & Sidebars** menu.)

By default, the macro's name is the same as the command (%(Python) "%D/%f"). To change the name to something more friendly, right-click on the macro and select **Properties**.

Specifying Parameters

- **Specify as part of Run command**

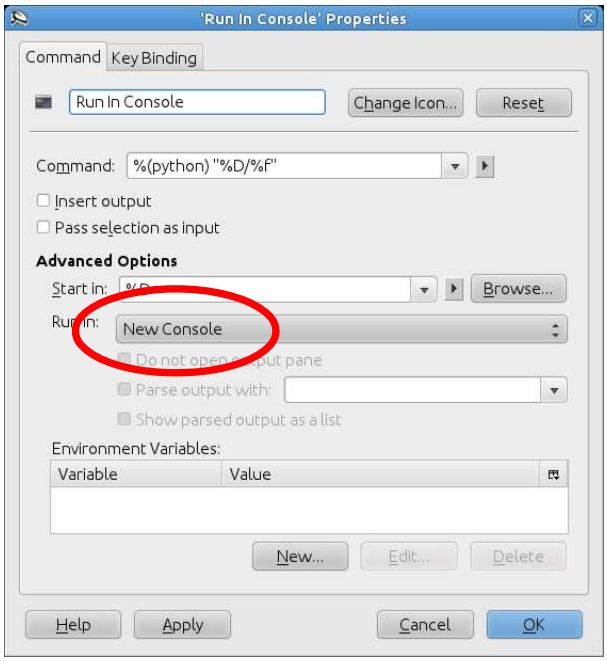
If you need command line parameters for your script, add them as part of the Run command:



Running in a console window

- **Better for interactive programs (using raw_input)**

If your program uses raw_input(), it is better to run it in a separate console window. To do this, follow the procedure described for creating a command, but under "Run In:", select New Console.



Appendix E

Using sympy

About sympy

- **Library for symbolic math**
- **Does not require external libraries**
- **Features**
  - **Basic arithmetic**
  - **Arbitrary precision integers, rationals, and floats**
  - **Polynomials**
  - **Calculus**
  - **Solving Equations**
  - **Combinatoricsx**
  - **Discrete math**
  - **Matrices**
  - **Geometric Algebra**
  - **Geometry**
  - **Plotting**
  - **Physics**
  - **Statistics**
  - **Printing**

Sympy is a package for creating and manipulating mathematical expressions. It contains many subpackages for all areas of symbolic logic.

from the SymPy documentation: *SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python and does not require any external libraries.*

Expressions

- **Objects of type sympy.Expr**
- **Symbols, numbers, etc. bound with operators**
- **SymPy will try to simplify**
- **SymPy will not simplify if any information would be lost**
- **Expressions are immutable**

Expressions consist of symbols, numbers, functions, and function applications, with operators binding them together. Expressions are Expr objects, and can themselves be manipulated with operators.

Example

sym_expressions.py

```
from sympy import Symbol, sin, pi

x = Symbol('x')
e1 = x + 1
print e1

y = Symbol('y')
e2 = x - y + 17 + y - 16 + sin(pi)
print e2

print e1 + e2
print e1 + e2 + 4 * x**2
print e1**2
```

```
sym_expressions.py
1 + x
1 + x
2 + 2*x
2 + 2*x + 4*x**2
(1 + x)**2
```

Built-in Numeric Types

- **Integer**
- **Float**
- **Rational**

sympy defines three main numeric types: Integer, Float, and Rational.

The Rational class represents a rational number as an (Integer) numerator/denominator pair of two Integers: . So Rational(1,2) represents 1/2, Rational(5,2) represents 5/2, and so on.

As a shortcut sympy will convert native Python numbers into sympy numbers.

Example

sym_builtins.py

```
from sympy import Integer, Rational

a = Rational(2)
b = Rational(10)

result = a**50/b**50

print result

x = Integer(1000)
y = Integer(28000203)
result = x * y
print type(result)
print result
```

Symbols

- **Building blocks of expressions**
- **Not the same as builtin Python types**
- **Create with Symbol(), symbols(), or var()**
- **Common symbols predefined**

Before you can create expressions, you need symbols to use in the expressions. These symbols are objects that represent values.

You can create symbols with the Symbol() constructor. Pass the name of the symbol as a string. You can also use the convenience functions symbols() and var(). The difference is that var() puts the symbols in the namespace.

Predefined symbols can be imported from the sympy.abc package.

Note that the Python variable name does not necessarily have to match the sympy algebraic symbol name. For instance, in the example below, the Python variables **i** and **j** represent the algebraic symbols **a** and **b**.

Once declared, symbols maybe be reused.

As a convention, a symbol named "x_1" means subscript (x_1) and "x^1" means superscript (x^1)

Example

sym_symbols.py

```
from sympy import Symbol, var, symbols
from sympy.abc import x,y, theta

k = Symbol('k')
e1 = 2 * k + 1
print e1

e2 = x * y * theta
print e2

(i,j) = symbols('a b')
e3 = i**2 + j
print e3

var('m n')
e4 = 1/m**n
print e4
```

```
sym_symbols.py
1 + 2*k
theta*x*y
b + a**2
m**(-n)
```

Expression Terms

- **_args gives in fixed order if significant**
- **as_order_terms() always ordered**
- **Gives individual terms**

To get the component parts of an expression, use expression._args. This returns a tuple of the components. If the order is signifcant, they will be returned in order; otherwise they will not necessarily be.

If you need the components in order, use expression.as_ordered_terms().

Example

sym_terms.py

```
from sympy import Derivative, sin
from sympy.abc import x

e = Derivative(sin(x), x, x)
print e

print e._args
```

```
sym_terms.py
D(sin(x), x, x)
(sin(x), x, x)
```

Prettyprinting

- **Makes expressions look more normal**
- **Default printer puts everything one line**
- **import pprint function**
- **Optionally set use_unicode to True**

By default, sympy prints out expressions on one line. This can ake complex expressions hard to read. To output better versions, use the pprint() function.

Example

sym_pprint.py

```
import sys
from sympy import Symbol, var, symbols, pprint
from sympy.abc import x,y, theta

k = Symbol('k')
e1 = 2 * k + 1
pprint(e1)

e2 = x * y * theta
pprint(e2)

(i,j) = symbols('a b')
e3 = i**2 + j
pprint(e3)

var('m n')
e4 = 1/m**n
pprint(e4, use_unicode=True)
```

sym_pprint.py

```
1 + 2·k
θ·x·y
     2
b + a
 -n
m
```

If the supporting packages are installed, use preview() to pop up a window with an expression rendered in LaTeX

Numerical Computing

- **Basic use of sympy**
- **Use .evalf() method to evaluate function**
- **.evalf(precision)**
- **solve(expr) solves function**

One of the most import uses of sympy is for basic numerical computing. Once you have created an expression, use the .evalf() method to evaluate it. You can specify the precision of the result by passing a numeric value to evalf(). You can also pass in a dictionary, where the keys are the symbol names and the values are the corresponding values.

Example

sym_numeric.py

```
from sympy import pi, Symbol, symbols, log, pprint, solve

print pi.evalf(50)
print

x = Symbol('x')
f = x ** (1 - log(log(log(log( 1 / x )))))
pprint(f)

pprint(f.subs(x, pi).evalf())
pprint(f.subs(x, 10).evalf())
pprint(f.subs(x, 3).evalf())
print

a,b,c = symbols('a b c')
exp = (a+b)*40-(c-a)/0.5
print exp
print exp.evalf(6,subs={a:6, b:5, c:2})
```

sym_numeric.py

```
3.1415926535897932384626433832795028841971693993751

                ⎛    ⎛    ⎛    ⎛1⎞ ⎞ ⎞ ⎞
 1 - log⎜log⎜log⎜log⎜─⎟ ⎟ ⎟ ⎟
                ⎝    ⎝    ⎝    ⎝x⎠ ⎠ ⎠ ⎠
x
0.730290019485505 - 1.30803618190213⋅ⅈ
-0.95457210781002 - 2.53547071152406⋅ⅈ
0.762474348496821 - 1.25129887750605⋅ⅈ

40*b + 42.0*a - 2.0*c
448.000
```

7400 East Orchard Road, Suite 1450N
Greenwood Village, Colorado 80111
Ph: 303-302-5280 FX: 303-302-5281
www.itcourseware.com

9-35-00035-000-06-30-14