

Grad Refresher Series 2011

Python

Marius Muja
mariusm@cs.ubc.ca

September 28, 2011

The Python Programming Language

- ▶ Interpreted - but can have modules written in C/C++
- ▶ Dynamically, strongly typed
 - ▶ dynamically typed - no need to declare variable types
 - ▶ strongly typed - restrictions on how variable of different types may be combined
- ▶ Object Oriented - everything is an object
- ▶ Allows multiple programming paradigms: OO, procedural, functional

Why use Python?

- ▶ Want a language easy to learn
 - ▶ Clean, clear syntax
 - ▶ Very few keywords
- ▶ Want to write a program fast
 - ▶ programs 2-10x shorter than the C,C++,Java equivalent
- ▶ Want to be able to read your code next year
 - ▶ ... or somebody else to read your code
- ▶ You think that “batteries included” is a good idea
 - ▶ it has an extensive standard library
 - ▶ 3rd party libraries available for just about anything
- ▶ *Because it makes programming fun!*

Python Programs

- ▶ Text files, traditionally with a .py extension
 - ▶ .pyc and .pyo automatically generated when you run the program
- ▶ Programs vs Modules
 - ▶ a .py file can be program or a module
 - ▶ it's a program when executed directly

```
$ python hello.py
```

- ▶ it's a module when referenced via the `import` statement

```
import hello
```

Programs and Modules

- ▶ `__name__` variable used to distinguish between the two
- ▶ useful for regression testing
 - ▶ when executed as a program the test is executed
 - ▶ when imported as a module the test is not executed

```
if __name__ == "__main__":  
    run_test()
```

Variables

- ▶ Variables need no declaration

```
>>> a=1
>>>
```

- ▶ Variables must be created before they can be used

```
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>>
```

Types

- ▶ Everything has a type

```
>>> a=1
>>> type(a)
<type 'int'>
>>> b='2'
>>> type(b)
<type 'str'>
>>> type(1.0)
<type 'float'>
```

- ▶ Strong typing

```
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Simple types

- ▶ Strings

- ▶ May hold any data (including NULLs)

```
>>> s = 'Hello\nPython!'
>>> print s
Hello
Python!
```

- ▶ Raw strings

```
>>> s = r"Hello\nPython!"
>>> print s
Hello\nPython!
```

- ▶ Multi-line strings

```
>>> s = """Hello
Python!"""
>>> print s
Hello
Python!
```


Simple types

- ▶ Integer - implemented using C longs
 - ▶ Like in C, division returns floor

```
>>> 7 / 2  
3
```

- ▶ Long - have unlimited size

```
>>> 2**500  
327339060789614187001318969682759915221664204604306  
478948329136809613379640467455488327009232590415715  
0886684127560071009217256545885393053328527589376L
```

- ▶ Float - implemented using C doubles

High level types

- ▶ Lists
 - ▶ Hold a sequence of items
- ▶ Tuples
 - ▶ Similar to lists, but immutable
- ▶ Dictionaries
 - ▶ Hold key-value pairs

Lists

- ▶ Hold a sequence of items
- ▶ May hold any types

```
>>> li = []
>>> li.append(3.0)
>>> li.extend(['a', 'b', 'Python'])
>>> len(li)
4
>>> li
[3.0, 'a', 'b', 'Python']
>> li[1]
'a'
>> li.index('Python')
3
>>> li[-2]
'b'
```

Lists

- ▶ List slicing

```
>>> li[1:3]
['b', 'c']
['b', 'c']
>>> li[1:-1]
['b', 'c', 'd']
```

- ▶ List operators

```
>>> li = ['a','b']*3
>>> li
['a', 'b', 'a', 'b', 'a', 'b']
>>> li = ['a','b']+['c','d']
>>> li
['a', 'b', 'c', 'd']
```

List comprehensions

- ▶ provide a concise way to create lists

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
```

Tuples

- ▶ Similar to lists, but immutable
- ▶ Often used in place of simple structures

```
>>> point = (3,4)
>>> point
(3, 4)
```

- ▶ Automatic unpacking

```
>>> x,y = point
>>> x
3
```

- ▶ Used to return multiple values from functions

```
>>> x,y = GetPoint()
```

Dictionaries

- ▶ Hold key-value pairs
- ▶ Often called maps, or hash tables
- ▶ Keys may be any immutable objects, values may be any objects

```
>>> d = {}
>>> d[1] = "Python"
>>> d["hello"] = 1.0
>>> d[(1,2)] = 3
>>> d
{(1, 2): 3, 1: 'Python', 'hello': 1.0}
>>> d['hello']
1.0
>>> len(d)
3
```

Blocks

- ▶ Blocks are delimited by indentation
- ▶ Colon used to start a new block

```
if a>0:  
    print "Computing_square_root"  
    b = sqrt(a)
```

- ▶ Many hate this when they first see it
- ▶ Python programmers come to love it
- ▶ Code gets to be more readable
 - ▶ Humans use indentation anyway when reading code to determine its structure
 - ▶ Ever got bitten by the C code:

```
if (a>0)  
    printf("Computing_square_root\n");  
    b = sqrt(a);
```


Conditionals

- ▶ if, elif, else

```
if condition:  
    [block]  
elif condition:  
    [block]  
else:  
    [block]
```

Looping

- ▶ For loop

```
for el in iterable:  
    [block]
```

- ▶ The classic for loop

```
for i in range(100):  
    print i
```

- ▶ While loop

```
while condition:  
    [block]
```

- ▶ break, continue - the usual operation

Looping techniques

- ▶ Looping through dictionaries

```
>>> knights = {'gallahad': 'the_pure', 'robin': 'the_brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

- ▶ Using position and values inside the loop

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Looping techniques

- ▶ Looping through multiple sequences at a time

```
>>> questions = ['name', 'quest', 'favorite_color']
>>> answers = ['lancelot', 'the_holy_grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What_is_your_{0}?_It_is_{1}'.format(q, a)
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

Functions

- ▶ Function declaration

```
def function_name(argument_list):  
    [block]
```

- ▶ Function arguments can have default values

```
def print_elements(sequence, sep = "_"):  
    print sep.join([str(k) for k in sequence])
```

- ▶ Functions are objects too

- ▶ Can be passed to other functions, assigned to variables,...

```
>>> print_elements  
<function print_elements at 0xb7df4aac>  
>>> print_sequence = print_elements  
>>> type(print_sequence)  
<type 'function'>  
>>> print_sequence([1,2,3.0], '-')  
1-2-3.0
```

Functions

- ▶ Keyword arguments

```
>>> print_elements(sep=';', sequence=[1,2,3])  
1;2;3
```

- ▶ Variable number of arguments

```
def print_arguments(*arguments, **keyword_args):  
    print "Positional arguments:", arguments  
    print "Keyword arguments:", keyword_args
```

```
>>> print_arguments(1,2,3, sep='-', list=['a','b','c'], count=10)  
Positional arguments: (1, 2, 3)  
Keyword arguments: {'count': 10, 'list': ['a', 'b', 'c'], 'sep': '-'}
```

Classes

- ▶ Declaration

```
class ClassName(BaseClass):  
    [block]
```

- ▶ Example

```
class Point:  
    def __init__(self,x,y):  
        self.x = x  
        self.y = y  
  
    def getPoint(self):  
        return (self.x,self.y)
```

Classes

- ▶ Classes are objects too
 - ▶ Can be passed to functions, assigned to variables

```
>>> Point
<class __main__.Point at 0xb79e8cec>
```

- ▶ Classes get instantiated using call syntax

```
>>> p = Point(1,2)
>>> p
<__main__.Point instance at 0xb77734ec>
>>> p.x
1
>>> p.y
2
```


Classes

- ▶ The constructor has a special name: `__init__`
 - ▶ the destructor is called `__del__`
- ▶ The `self` parameter is the instance
 - ▶ similar to `this` from C++
 - ▶ it's explicit in Python, implicit in C++
 - ▶ the name `self` is just a convention, it can be called anything

Modules

- ▶ Each module has its own namespace
- ▶ Modules can be implemented in either Python or C/C++
- ▶ `import` statement makes a module visible

```
>>> import math
>>> math.sin(1.2)
0.93203908596722629
>>> sin(1.2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> from math import sin
>>> sin(1.2)
0.93203908596722629
>>> from math import *           # import everything from module math
```

Docstrings

- ▶ convenient way of including documentation with the code

```
class Point:
    """Python class that models a point"""

    def __init__(self, x, y):
        """Class constructor"""
        self.x = x
        self.y = y

    def getPoint(self):
        """Returns point as a tuple (x,y)"""
        return (self.x, self.y)
```

```
>>> p = Point(1,2)
>>> p
<docstring.Point instance at 0xa1c0a8c>
>>> p.__doc__
'Python_class_that_models_a_point'
>>> p.getPoint.__doc__
'Returns_point_as_a_tuple_(x,y)'
```

Example

```
# file shape.py

class Shape:
    # constructor
    def __init__(self, initx, inity):
        self.moveTo(initx, inity)

    # accessors for x & y
    def getX(self):
        return self.x
    def getY(self):
        return self.y
    def setX(self, newx):
        self.x = newx
    def setY(self, newy):
        self.y = newy

    # move the shape position
    def moveTo(self, newx, newy):
        self.setX(newx)
        self.setY(newy)
    def moveBy(self, deltax, deltay):
        self.moveTo(self.getX() + deltax, self.getY() + deltay)

    # abstract draw method
    def draw(self):
        pass
```

Example

```
# file rectangle.py

from shape import Shape

class Rectangle(Shape):

    # constructor
    def __init__(self, initx, inity, initwidth, initheight):
        Shape.__init__(self, initx, inity)
        self.setWidth(initwidth)
        self.setHeight(initheight)

    # accessors for width & height
    def getWidth(self):
        return self.width
    def getHeight(self):
        return self.height
    def setWidth(self, newwidth):
        self.width = newwidth
    def setHeight(self, newheight):
        self.height = newheight

    # draw the rectangle
    def draw(self):
        print "Drawing_a_Rectangle_at:(%d,%d),_width_%d,_height_%d" % \
            (self.getX(), self.getY(), self.getWidth(), self.getHeight())
```

Example

```
# file circle.py
from shape import Shape

class Circle(Shape):

    # constructor
    def __init__(self, initx, inity, iniradius):
        Shape.__init__(self, initx, inity)
        self.setRadius(iniradius)

    # accessors for the radius
    def getRadius(self):
        return self.radius
    def setRadius(self, newradius):
        self.radius = newradius

    # draw the circle
    def draw(self):
        print "Drawing_a_Circle_at:(%d,%d),_radius_%d" % \
            (self.getX(), self.getY(), self.getRadius())
```

Example

```
# file test_shapes.py

from rectangle import Rectangle
from circle import Circle

def test_shapes():

    # set up lists to hold the shapes
    scribble = [Rectangle(10, 20, 5, 6), Circle(15, 25, 8)]

    # iterate through the lists and handle shapes polymorphically
    for each in scribble:
        each.draw()
        each.moveBy(100, 100)
        each.draw()

    # call a rectangle specific instance
    arec = Rectangle(0, 0, 15, 15)
    arec.setWidth(30)
    arec.draw()

if __name__ == "__main__":
    test_shapes()
```

```
$ python test_shapes.py
Drawing a Rectangle at:(10,20), width 5, height 6
Drawing a Rectangle at:(110,120), width 5, height 6
Drawing a Circle at:(15,25), radius 8
Drawing a Circle at:(115,125), radius 8
Drawing a Rectangle at:(0,0), width 30, height 15
```

Integrating Python and C/C++

- ▶ There are multiple ways of combining Python code with C/C++ code
- ▶ Typical workflows:
 - ▶ Have a C/C++ library and want to access it from Python
 - ▶ Have a Python application and want to speed it up by rewriting parts of it in C/C++
 - ▶ Use Python as a “glue language” to combine different pieces of C/C++ code

Integrating Python and C/C++

- ▶ boost::python
 - ▶ Creates a Python module directly from C++. Part of the C++ boost library. Very simple to use if you are already familiar to C++.
- ▶ ctypes
 - ▶ Call functions from DLLs or shared libraries from Python. Part of the Python standard library.
- ▶ cython/pyrex
 - ▶ Language for writing Python extensions easy, mix between python and C.
- ▶ pyInline, scipy::weave
 - ▶ C/C++ code fragments in the python source code
- ▶ Others...
 - ▶ Py++, SWIG, SIP

Python for scientific computing

- ▶ Many, many libraries
 - ▶ http://www.scipy.org/Topical_Software
- ▶ NumPy
 - ▶ the fundamental package needed for scientific computing with Python
- ▶ SciPy
 - ▶ variety of high level science and engineering modules together as a single package
- ▶ matplotlib
 - ▶ 2D plotting library which produces publication quality figures
- ▶ mlabwrap
 - ▶ high-level python to Matlab bridge that lets Matlab look like a normal python library

Numpy

- ▶ the fundamental package needed for scientific computing with Python
- ▶ a powerful N-dimensional array object
- ▶ tools for integrating C/C++ and Fortran code
- ▶ useful linear algebra, Fourier transform, random numbers capabilities
- ▶ http://www.scipy.org/NumPy_for_Matlab_Users

Numpy

► A taste of NumPy

```
>>> from numpy import *
>>> a = array( [ 10, 20, 30, 40 ] ) # create an array out of a list
>>> a
array([10, 20, 30, 40])
>>> b = arange( 4 ) # create an array of 4 integers, from 0 to 3
>>> b
array([0, 1, 2, 3])
>>> d = a+b**2 # elementwise operations
>>> d
array([10, 21, 34, 49])
```

```
>>> x = ones( (3,4) )
>>> x
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
>>> x.shape # a tuple with the dimensions
(3, 4)
>>> a = random.normal(0,1,(3,4)) # random samples from a normal distribution
>>> a
array([[ -1.20183817,  0.03338838,  1.09723418, -0.08546884],
       [-0.74220878,  0.34840145, -0.42426146, -0.46312178],
       [ 0.39493244,  1.78215556,  0.39265006, -0.45922891]])
```

Numpy

- ▶ http://www.scipy.org/Numpy_Example_List
- ▶ Example: `svd()`

```
>>> from numpy import *
>>> from numpy.linalg import svd
>>> A = array([[1., 3., 5.],[2., 4., 6.]]) # A is a (2x3) matrix
>>> U,sigma,V = svd(A)
>>> print U # U is a (2x2) unitary matrix
[[-0.61962948 -0.78489445]
 [ -0.78489445  0.61962948]]
>>> print sigma # non-zero diagonal elements of Sigma
[ 9.52551809  0.51430058]
>>> print V # V is a (3x3) unitary matrix
[[-0.2298477 -0.52474482 -0.81964194]
 [ 0.88346102  0.24078249 -0.40189603]
 [ 0.40824829 -0.81649658  0.40824829]]
>>> Sigma = zeros_like(A) # constructing Sigma from sigma
>>> n = min(A.shape)
>>> Sigma[:n,:n] = diag(sigma)
>>> print dot(U,dot(Sigma,V)) # A = U * Sigma * V
[[ 1.  3.  5.]
 [ 2.  4.  6.]
```

Matplotlib

- ▶ Powerful 2D plotting library (also 3D plotting with the mplot3d toolkit)
- ▶ Provides a set of functions familiar to MATLAB users as well as an object oriented interface
- ▶ Example

```
#!/usr/bin/env python
from pylab import *

mu, sigma = 100, 15
x = mu + sigma*randn(10000)

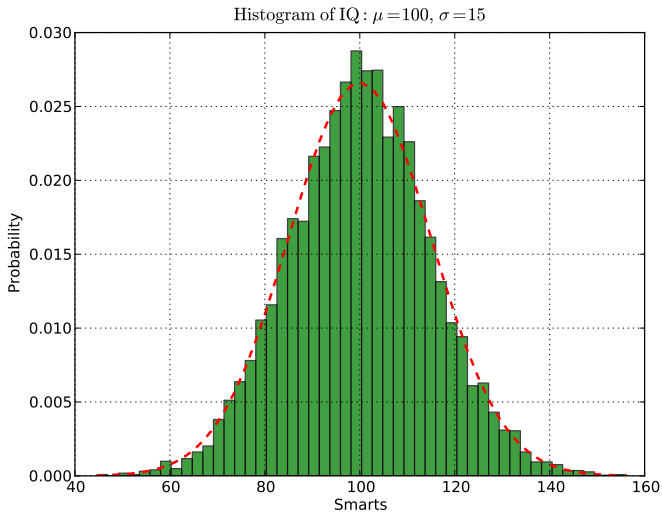
# the histogram of the data
n, bins, patches = hist(x, 50, normed=1, facecolor='green', alpha=0.75)

# add a 'best fit' line
y = normpdf( bins, mu, sigma)
l = plot(bins, y, 'r—', linewidth=1)

xlabel('Smarts')
ylabel('Probability')
title(r'$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
axis([40, 160, 0, 0.03])
grid(True)
show()
```

Matplotlib

► Example



Scipy

- ▶ project which includes a variety of high level science and engineering modules together as a single package
 - ▶ linear algebra (including wrappers to BLAS and LAPACK)
 - ▶ optimization
 - ▶ integration
 - ▶ special functions
 - ▶ FFTs
 - ▶ signal and image processing
 - ▶ genetic algorithms
 - ▶ ODE solvers
 - ▶ others...

Scipy

► Example

```
from numpy import *
from numpy.random import randn

x = arange(0,6e-2,6e-2/30)
A,k,theta = 10, 1.0/3e-2, pi/6
y_true = A*sin(2*pi*k*x+theta)
y_meas = y_true + 2*randn(len(x))

def residuals(p, y, x):
    A,k,theta = p
    err = y-A*sin(2*pi*k*x+theta)
    return err

def peval(x, p):
    return p[0]*sin(2*pi*p[1]*x+p[2])

p0 = [8, 1/2.3e-2, pi/3]

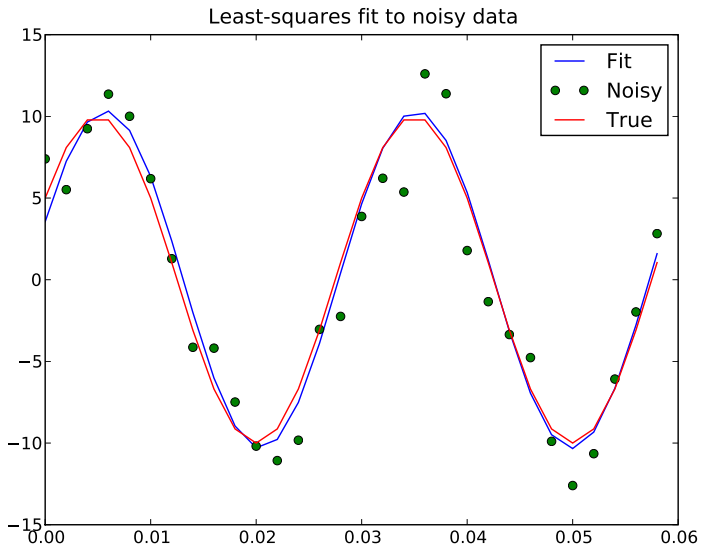
from scipy.optimize import leastsq
plsq = leastsq(residuals, p0, args=(y_meas, x))

# plotting
from pylab import *
clf()
plot(x,peval(x,plsq[0]),x,y_meas,'o',x,y_true)
title('Least-squares_fit_to_noisy_data')
legend(['Fit', 'Noisy', 'True'])

savefig('fig.pdf')
```

Scipy

▶ Example



Scipy

► Subpackages

```
>>> import scipy
>>> help(scipy)
...
odr — Orthogonal Distance Regression
misc — Various utilities that don't have another home.
sparse.linalg.eigen.arpack — Eigenvalue solver using iterative methods
fftpack — Discrete Fourier Transform algorithms
io — Data input and output
sparse.linalg.eigen.lobpcg — Locally Optimal Block Preconditioned
Conjugate Gradient Method (LOBPCG)
special — Airy Functions
lib.blas — Wrappers to BLAS library
sparse.linalg.eigen — Sparse Eigenvalue Solvers
stats — Statistical Functions
lib — Python wrappers to external libraries
optimize — Optimization Tools
maxentropy — Routines for fitting maximum entropy models
integrate — Integration routines
ndimage — n-dimensional image package
linalg — Linear algebra routines
spatial — Spatial data structures and algorithms
interpolate — Interpolation Tools
sparse.linalg — Sparse Linear Algebra
sparse.linalg.dsolve.umfpack — :Interface to the UMFPACK library:
sparse.linalg.dsolve — Linear Solvers
lib.lapack — Wrappers to LAPACK library
cluster — Vector Quantization / Kmeans
signal — Signal Processing Tools
sparse — Sparse Matrices
...
```

SageMath

- ▶ <http://www.sagemath.org/>
- ▶ free open-source mathematics software system
- ▶ combines the power of many existing open-source packages into a common Python-based interface
- ▶ Mission: Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab.
- ▶ Can be used on the department machines:

```
$ use sage
```

Questions?