py2: Dive into Python
What Programming Used to Be Like
Hundreds of Programming Languages

• Several ways of categorization
  – paradigm
    • imperative vs functional
    • procedural, object-oriented, generic, etc.
  – data model
    • memory and address
    • object-oriented
Memory and Address

• Low-level programming languages
  – Machine Code
  – Assembly

• Mid-level programming languages
  – C
  – C++
int func(double a) {
    int c;
    c = a + 1;
    return c;
}

b = func(1.1);
Symbol and Value

• High Level Programming Languages
  – Strongly typed
    • FORTRAN
    • C++
    • Matlab
  – Weakly typed
    • Python
    • TCL
Python Data Model

• Object-oriented
  – everything is object
• variable are name-tags
  – assignment : attach a tag to the object
• garbage collection
  – object with no tag is garbage
• containers (list, tuple, dict)
  – mutable and immutable objects
def func(a):
    c = a + 1
    return c

...  

b = func(1.1)

Python: Symbol and Value

Before calling
def func(a):
    c = a + 1
    return c

... 

b = func(1.1)
def func(a):
    c = a + 1
    return c

... 

b = func(1.1)

After returning

1.1

2.1

b
def func(a):
    c = a + 1
    return c

... 

b = func(1.1)

c = b*2.0
def func(a):
    c = a + 1
    return c

... 

b = func(1.1)

c = b*2.0

b = "hello"
List in Python: a Container

\[ a = [ 1, "a", 1.3 ] \]
List in Python: Mutable

```python
a = [1, "haha", 1.3]
a[1] = a[0] + a[2]
a.append(100)
```
Tuple: Immutable

\[ b = (1, \text{"haha"}, 1.3) \]


\[ b.append(100) \]
Tuple: Immutable

```
b = ( 1, "haha", 1.3 )
b = ( b[0], a[0] + a[2], b[2], 100 )
```
Dict in Python: key-value pairs

\[ r = \{ \text{"name"} : \text{"Bob"}, \text{"score"} : 61, \text{"pass"} : \text{True} \} \]
Dict in Python : accessing by key

```python
r = {“name”:“Bob”, “score”:61, “pass”:True }

print r[“name”]       # get value
r[“score”] = 59       # set value
r[100] = 3.14         # insert item
r.pop(“score”)        # remove item

r.has_key(“name”)     # True
r.has_key(“score”)    # False

r.keys()              # [“name”, “pass”, 100]
r.values()            # [“Bob”, False, 3.14]
```
Everything is Object

• Primitives
  – integer, float, string, bool, None, ...

• Containers
  – list, tuple, dict, ...

• Code
  – function, class, module

• Instance of user-defined class
from math import sin

def f(x):  # define function object
    return sin(x)

def g(func, x):
    return func(x)+func(x+3.14/2)

for x in [0.0, 0.1, 0.2, 0.3]:
    print g(f, x)  # pass function object
class Vector(object):  # define class object
    def __init__(self, x, y, z):  # constructor
        self.x = x
        self.y = y
        self.z = z
    def size(self):  # method
        return sqrt(self.x*self.x +
                    self.y*self.y + self.z*self.z)

vx = Vector(1, 0, 0)  # instance object
vy = Vector(0, 1, 0)
vx.size()

Vector.__dict__.keys()  # "__init__", "size", ...
vx.__dict__.keys()  # "x", "y", "z"
Vector.__dict__[‘size’](vx)  # vx.size()
Class as a Container

```
Vector.__dict__

{ _:_, _:_, _:_, _:_, ... }

__(self, x, y, z):
    self.x = x
    self.y = y
    self.z = z

__(self):
    return sqrt(x*x+y*y+z*z)
```
Object-Oriented Paradigm

- Encapsulation
- Abstraction/Specialization
- Inheritance
- Polymorphism
- Delegation
- “Duck-typing”
start

input initial guess \( x^{(0)} \)

compute residue \( e^{(n)} = F(x^{(n)}) \)

\( e^{(n)} \leq \varepsilon \) ?

Y

output \( x^{(n)} \)

stop

N

compute Jacobian \( J(x^{(n)}) \)

\( x^{(n+1)} = x^{(n)} - J^{-1} e^{(n)} \)
Encapsulation: Solution State

class NLEqnState(object):
    def __init__(self, n=0):
        self.N = n                      #n: number of eqns/vars
        self.x = None                   #solution of current iteration
        self.J = None                   #Jacobian of current iteration
        self.dx = None                  #update to be applied

        if n>0:
            self._prepareData()

    def _prepareData(self):
        self.x = scipy.zeros(self.N)
        self.b = scipy.zeros(self.N)
        self.dx = None
Abstraction: Nonlinear Eqn

```python
class NLEqns(object):
    def __init__(self):
        self.state = None

    def calcFunJac(self):
        pass

    def initGuess(self):
        pass

    def checkConv(self):
        ...

    def solve(self):
        ...
```

# Nonlinear Eqn class

# empty state

# we know nothing abt
# the eqns yet

# nor do we know this

# some default criteria

# Newtons Method here!
def solve(self):
    maxiter=50             # max number of iterations
    for iter in xrange(0,maxiter):
        self.state.clearFunJac()
        self.calcFunJac()     # Calculate Function & Jacobian
        flagConv, err = self.checkConv()
        if flagConv:          # converged?
            break

# Newton update vector
    self.state.dx = np.negative(dsolve.spsolve(
                                        self.state.J.tocsr(), self.state.b))

# solution in the next iteration
    self.state.x = np.add(self.state.x, self.state.dx)
Specialization: Our Sample Eqn

class SimpleEqns(NLEqns):
    def __init__(self):
        super(SimpleEqns, self).__init__()
        # prepare solution state data for 2 unknowns
        self.state = NLEqnState(2)

    def initGuess(self):
        # initial guess
        self.state.x = np.array([1, 1])

    def calcFunJac(self):
        ...
def calcFunJac(self):
    # get solution of the last iteration
    x, y = self.state.getVars([0,1])

    # compute function, and partial derivative
    # information is contained in f0 and f1.
    # black magic!!! we'll explain later
    f0 = x*x + y*y
    f1 = x*x - y*y

    # set function value as well as
    # jacobian matrix elements
    self.state.setFunJac(0, f0)
    self.state.setFunJac(1, f1)
Inherited solve() Method

# main program
eqn = SimpleEqns()
eqn.initGuess()

# solve with newton’s method
# the solve() method is inherited from NLEqn class
eqn.solve()

print 'Result', eqn.state.x
def solve(self):
    maxiter=50
    # max number of iterations
    for iter in xrange(0,maxiter):
        self.state.clearFunJac()
        self.calcFunJac()  # Calculate Function & Jacobian
        flagConv, err = self.checkConv()
        if flagConv:
            break

    # Newton update vector
    self.state.dx = np.negative(dsolve.spsolve(
        self.state.J.tocsr(), self.state.b))

    # solution in the next iteration
    self.state.x = np.add(self.state.x, self.state.dx)
Duck Typing

If it quacks like a duck,
and walk like a duck,
and .... like a duck,
...
then for all practical purposes,
it is a duck.
class Vector(object):
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __add__(self, other):
        v = Vector(self.x+other.x,
                   self.y+other.y,
                   self.z+other.z)
        return v

a = Vector(1,0,0)
b = Vector(0,1,0)
print a+b  # (1,1,0)