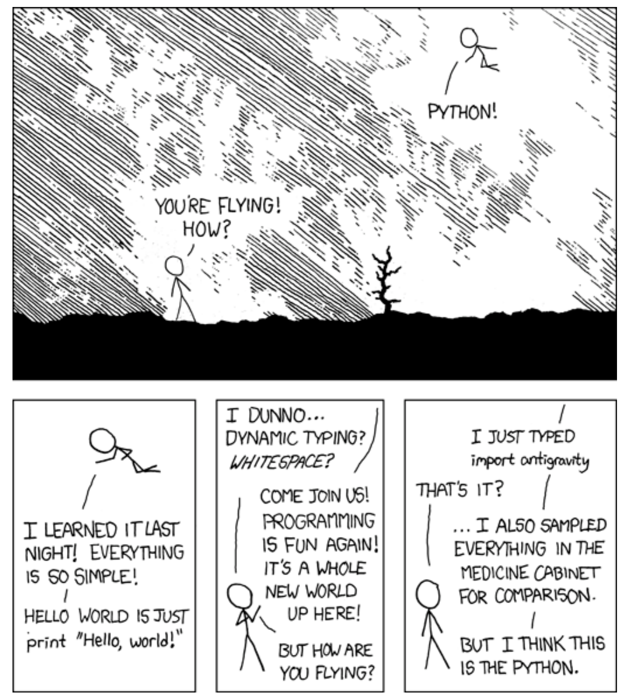
Python and Numpy

What is Python?

- Fast code development and test
- A dynamically typed language
  - You do not need to declare the type of a variable
- Syntax is very much C-like
- A huge number of packages for doing almost everything (numpy, scipy, matplotlib, …)
- Automatic memory management
- Functions can be passed around as arguments
- Can bind to C code easily for speed



(from Jonathan Woodring, Los Alamos NL)

Variable Assignment

```
# assignment & print function

x = 0 # you don't have to declare x first
print(x) # calling built-in function

y = 1.0 # just assign a literal to a name
print(y) # calling looks just like C and other Algol-languages

z = 'foo' # a string with quotes
print(z) # no semi-colons or other terminators

w = "bar" # another string with double quotes (either works)
print(w)
```

```
0
1.0
foo
bar
```

More Assignment

```
# more assignment & expressions

x = 4
y = 2
z = 3
r = 'one'
s = 'two'
t = 'three'
f = 1.0

a = x * y + z # expressions look like most other infix notation
print(a)

print(r + s + t) # concatenating strings & expression
                 # in a function argument

b, c, d = x + f, 4 * f, y ** z # multiple assigments on one line
print(b, c, d) # adding numeric types casts integer to float

print(x + r) # but this doesn't work
             # (won't cast a numeric to a string, unlike Javascript)
```

```
11
onetwothree
(5.0, 4.0, 8)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-5f6b17d95c9a> in <module>()
```

Lists

```
# lists (vectors, really)

x = [1, 2, 3, 4, 5] # a list of integers
print(x)

y = [1.0, 2.0, 3.0, 4.0, 5.0] # a list of floating point
print(y)

z = ['a', 'b', 'c', 'd', 'e'] # a list of strings
print(z)

w = [1, 'two', 3.0, "four", print, 'last'] # can we mix them?
print(w) # yes, we can
```

```
[1, 2, 3, 4, 5]
[1.0, 2.0, 3.0, 4.0, 5.0]
['a', 'b', 'c', 'd', 'e']
[1, 'two', 3.0, 'four', <built-in function print>, 'last']
```

Slicing Lists

```
# list slices

w = [1, 2, 3, 4, 5]

# slices - like Matlab and Fortran
print(w[2:]) # everything from 2 onwards
print(w[:2]) # right hand index is exclusive
print(w[:2] + w[2:]) # list concatenation

# you can do subranges
print(w[1:3])

# you can do skips
print(w[::2])

# even in reverse
print(w[::-1])

# you can combine them all together
print(w[3:0:-2]) # notice I had to do 3 to 0 by -2 to go in reverse
```

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

Slicing Lists

```
# list slices

w = [1, 2, 3, 4, 5]

# slices - like Matlab and Fortran
print(w[2:]) # everything from 2 onwards
print(w[:2]) # right hand index is exclusive
print(w[:2] + w[2:]) # list concatenation

# you can do subranges
print(w[1:3])

# you can do skips
print(w[::2])

# even in reverse
print(w[::-1])

# you can combine them all together
print(w[3:0:-2]) # notice I had to do 3 to 0 by -2 to go in reverse
```

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

Iterating Lists with Loops

```
# list iteration

z = [1, 2, 3]
n = []
for i in z: # iterating over a list
    n.append(i + 1) # append is a method on a list
                    # that modifies it in place (it returns None)
print(n, z)

w = [[1, 2], [3, 4], [5, 6]]
s = ''
for i in w:
    for j in i:
        s = s + str(j)
print(s)

q = []
# iterate over two lists in tandem with zip
for i, j in zip(z, n):
    print('i:', i, 'j:', j, 'i+j:', i + j)
    q.append(i + j)
print(q)
```

```
[2, 3, 4] [1, 2, 3]
123456
i: 1 j: 2 i+j: 3
i: 2 j: 3 i+j: 5
i: 3 j: 4 i+j: 7
[3, 5, 7]
```

Iterating Lists with Loops

```python
# list iteration

z = [1, 2, 3]
n = []
for i in z: # iterating over a list
    n.append(i + 1) # append is a method on a list
                    # that modifies it in place (it returns None)
print(n, z)

w = [[1, 2], [3, 4], [5, 6]]
s = ''
for i in w:
    for j in i:
        s = s + str(j)
print(s)

q = []
# iterate over two lists in tandem with zip
for i, j in zip(z, n):
    print('i:', i, 'j:', j, 'i+j:', i + j)
    q.append(i + j)
print(q)
```

```
[2, 3, 4] [1, 2, 3]
123456
i: 1 j: 2 i+j: 3
i: 2 j: 3 i+j: 5
i: 3 j: 4 i+j: 7
[3, 5, 7]
```

If-Then-Else

```
# if-then-else & block indentation

x = 1
y = 2
z = 3

# see how blocks line up due to spacing?
# PEP 8 says the preferred tab stop is 4 spaces (don't use tabs)
# I prefer 2, personally
if x < 1 or False:
    print('not here')
elif y < 2 and True:
    print('not here either')
elif z > 0:
    print('we got here')
    if not x != y:
        print('nope')
    elif z == 3:
        print('here too')
        if z < y or y < x:
            print('not here either')
        else:
            print('we got all the way here')
            while z > x:
                z = z - 1
                if y > x:
                    y = y - x
                else:
                    y = y - 2
            while z >= y:
                z = z - 1
                if x > 0:
                    x = x + 1
    else:
        print('nada')
else:
    print('not gonna get here')
print(x, y, z)
```

```
we got here
here too
we got all the way here
4 -1 -2
```

Tuples are Immutable

```python
# tuples - basically, immutable lists
empty = ()
print(empty)
print(len(empty))

a = (1, 2, 3)
print(a)

print(a[0], a[-1], a[1:])

for i in a:
    print(i + 1)

a[0] = 'a' # this is going to fail, because tuples are immutable
# strings are immutable, too
# s = 'a string'
# s[0] = 'a' will fail
```

```
()
0
(1, 2, 3)
1 3 (2, 3)
2
3
4
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-348201c4d557> in <module>()
     12     print(i + 1)
```

Dictionaries

```
# dicts : maps, hashes, associative arrays

empty = {}
print(empty)
print(empty.keys()) # dicts have keys
print(empty.values()) # and values (it's a map)

a = {'one': 1, 2: 'two', 'print': print} # we can store all
                                         # sorts of things in a dict,
                                         # just like a list and tuple
print(a)
print(a['one']) # and we can use all sorts of keys
print(a[2])
print(a['print']) # even functions can be fetched
a['print']("hi there, I'm a function in a dict!") # call it

for k in a:
    print('key:', k, 'value:', a[k])

print('a key' in a) # boolean is a built-in type: True or False
a['a key'] # this is going to fail
```

```
{}
dict_keys([])
dict_values([])
{2: 'two', 'print': <built-in function print>, 'one': 1}
1
two
<built-in function print>
hi there, I'm a function in a dict!
key: 2 value: two
key: print value: <built-in function print>
key: one value: 1
False
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-14-2361f8fa98e9> in <module>()
```

Function Definitions

```
# arguments to functions and returning values

def xyz(x): # one argument, no types needed
    return x, x # this means it is returning a tuple

print(xyz(1))

def uvw(u, v): # two arguments
    return (u, v) # we can do it explicitly, too

print(uvw(1, 2))

def abc(a, b, c):
    return [a, b, c] # we can return lists

print(abc(1, 2, 3))
```

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

File I/O

```
# files

f = open('foo.txt', 'w')
f.write('hi there!\n')
f.close()

g = open('foo.txt', 'r')
s = g.read(10)
g.close()

print(s)
```

```
hi there!
```

NumPy

- The most popular package for scientific computing
  - Efficient N-dimensional arrays
  - Useful for linear algebra, data transformation etc.
- Get Numpy from http://www.scipy.org

```
# what's next? numpy

import numpy

A = numpy.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(A)

# OK, so what's so special about that compared to the list?
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

NumPy

- The most popular package for scientific computing
  - Efficient N-dimensional arrays
  - Useful for linear algebra, data transformation etc.
- Get Numpy from http://www.scipy.org

```
# numpy arrays are fast, almost C speed
# as long as you do "large amounts of work"

import time

AL = range(0, 1000000)
BL = range(0, 1000000)
CL = [0] * len(AL)

start = time.time()
for i in range(0, len(AL)):
    CL[i] = AL[i] + BL[i]
print(time.time() - start)

A = numpy.array(range(0, 1000000), numpy.int32)
B = numpy.array(range(0, 1000000), numpy.int32)

start = time.time()
C = A + B
print(time.time() - start)
```

```
3.7429728507995605
0.009889364242553711
```

NumPy Slicing

```
# numpy notation is similar to array slicing
# and Matlab and Fortran matrix notation

A = numpy.array(range(0, 10))

V = A[::2] # this is a view (shallow copy)
V[0] = -10 # slices are views in numpy
print(V, A)
B = A.copy() # this is a deep copy of A
B[0] = 0
print(B, A)

C = A[::2] + B[::2]
print(C)

C = A[1:9] * B[:8]
print(C)

C = A[1:-3] - B[2:-2]
print(C)

C = A / B[:5] # this is going to fail, because they aren't the same shape
```

```
[-10   2   4   6   8] [-10   1   2   3   4   5   6   7   8   9]
[0 1 2 3 4 5 6 7 8 9] [-10   1   2   3   4   5   6   7   8   9]
[-10   4   8  12  16]
[ 0  2  6 12 20 30 42 56]
[-1 -1 -1 -1 -1 -1]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-1de9057c7938> in <module>()
     20 print(C)
     21
---> 22 C = A / B[:5] # this is going to fail, because they aren't the same shape

ValueError: operands could not be broadcast together with shapes (10,) (5,)
```

NumPy Array Reshaping

```
# numpy also supports multi-dimensional arrays
# default memory layout is:
# C, row-major, right-most index varies fastest

A = numpy.array(range(0, 8))
A = numpy.reshape(A, (2, 2, 2)) # change the shape of an array
                                # the total size (elements) must be the same
print(A)

print(A[0,0,0]) # this is different from nested lists
print(A[1,1,1])

A = numpy.transpose(A, axes=[0,2,1]) # swap around axes
print(A)
```

```
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
0
7
[[[0 2]
  [1 3]]

 [[4 6]
  [5 7]]]
```

NumPy Array Broadcasting

```python
# numpy also supports "broadcasting"

A = numpy.array(range(0, 4))
A = numpy.reshape(A, (2, 2))

print(A) # a 2x2 matrix

A = A + 1 # 1 is added to all elements
print(A)

v = numpy.array([-1, 1]) # let's make a vector
v = numpy.reshape(v, (2, 1)) # a column vector
print(v)

A = A * v # v gets broadcast over the columns
print(A)

v = numpy.reshape(v, (1, 2)) # now it's a row vector
print(v)

A = A - v # v gets broadcast over the rows
print(A)
```

```
[[0 1]
 [2 3]]
[[1 2]
 [3 4]]
[[-1]
 [ 1]]
[[-1 -2]
 [ 3  4]]
[[-1  1]]
[[ 0 -3]
 [ 4  3]]
```

NumPy Linear Algebra

```
# and a lot of what you want is probably
# in the linear algebra
# http://docs.scipy.org/doc/numpy/reference/routines.linalg.html

from numpy.linalg import linalg # a submodule of a module

A = numpy.array([[0, 1], [2, 3]])
B = numpy.array([[0, -1], [1, 0]])

print(linalg.dot(A, B)) # matrix multiply
print(numpy.outer(A, B)) # outer product
print(linalg.qr(A)) # qr factorization
print(linalg.svd(A)) # SVD
print(linalg.eig(A)) # eigenvectors and values
print(linalg.inv(A)) # inverse of A
# etc.
```

```
[[ 1  0]
 [ 3 -2]]
[[ 0  0  0  0]
 [ 0 -1  1  0]
 [ 0 -2  2  0]
 [ 0 -3  3  0]]
(array([[ 0., -1.],
       [-1.,  0.]]), array([[-2., -3.],
       [ 0., -1.]]))
(array([[-0.22975292, -0.97324899],
       [-0.97324899,  0.22975292]]), array([ 3.70245917,  0.54018151]), array([[-0.52573111, -0.85065081],
       [ 0.85065081, -0.52573111]]))
(array([-0.56155281,  3.56155281]), array([[-0.87192821, -0.27032301],
       [ 0.48963374, -0.96276969]]))
[[-1.5  0.5]
 [ 1.   0. ]]
```

NumPy I/O

```
# getting raw binary data in and out of numpy

A = numpy.arange(0, 10, .5, numpy.float32)
print(A)

f = open('foo.bin', 'wb')
A.tofile(f) # just do a to file and it will dump it in C-order
f.close()

la = len(A)
A = None
print(A)

f = open('foo.bin', 'rb')
A = numpy.fromfile(f, numpy.float32, la) # to read back in
                                          # you have to specify type and number

f.close()

print(A)
```

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5  7.
  7.5  8.   8.5  9.   9.5]
None
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5  7.
  7.5  8.   8.5  9.   9.5]
```