Introduction to Programming in Python

August 25, 2014

Nicholas Cain, PhD

Scientist 1, Allen Institute for Brain Science

ALLEN INSTITUTE *for* BRAIN SCIENCE

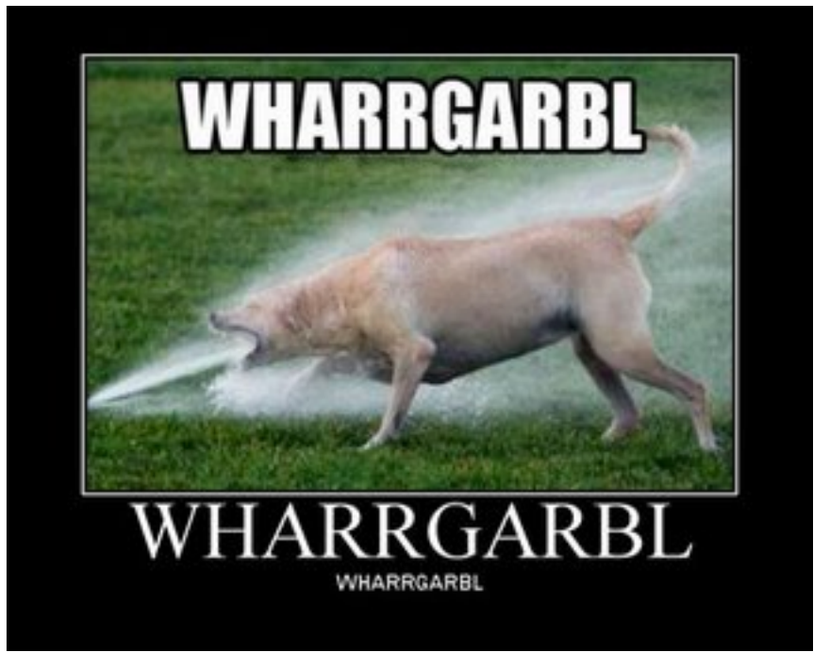
Fueling Discovery

Overview

- Background: What is Python?
- Syntax (lots of code examples)
- Modules? Packages?? Classes??? Oh my…
- Scientific programming in Python
- “Gotcha”s, i.e. what NOT to do
- Tips and Tricks, i.e. what TO DO

Questions for me:

- When would I use that?
- Why should I do that?
- What is that good for?



- What happens if I do *this* instead?
- How do I do *this*?
- Wait I totally don't get it. What are you saying?

Background: What is Python?

- Python is:
  - an object oriented programming language
  - intended to be flexible and readable
  - interpreted, as opposed to compiled
  - becoming a standard in tool in computational neuroscience

Background: Python in Neuroscience

PsychoPy
Psychology software in Python

neo

NEURON
for empirically-based simulations of neurons and networks of neurons

Nipype:
Neuroimaging in Python
Pipelines and Interfaces

stimfit
A program for viewing and analyzing electrophysiological data

nest::
simulated()

Nengo

BRIAN

pandas
$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

matplotlib

SciPy

ALLEN INSTITUTE
for BRAIN SCIENCE
Fueling Discovery

Nitime: time-series analysis
for neuroscience

NumPy

Getting started: (everything is free!)

- To start python from the command line, type:
  - “python” # Yes it is that easy…
  - “ipython” # better interactive prompt, same python
- For a more matlab-like interface:
  - Spyder: https://bitbucket.org/spyder-ide/spyderlib/downloads
- Eclipse IDE with pydev plugin is very powerful:
  - https://www.eclipse.org/downloads/

Getting started: (everything is free!)

- Linux/MacOSX: You probably already have it
- Windows: PythonXY
- You can have more than 1 (I have 4 on my laptop)
- Preferred install: anaconda
  - Numpy/Scipy, pip, and lots of other useful stuff
  - Free
  - Easy to install

Overview

- Background: What is Python?
- Syntax (lots of code examples)
- Modules? Packages?? Classes??? Oh my…
- Scientific programming in Python
- “Gotcha”s, i.e. what NOT to do
- Tips and Tricks, i.e. what TO DO

ALLEN INSTITUTE for BRAIN SCIENCE
Fueling Discovery

Syntax: Types

```
# Simple data types in python:
my_int = 42
my_float = 3.14159
my_string = "Hello World"
my_list = [1,2,3,'a']
my_tuple = (1,2,4)
my_dictionary = {'a':1, ‘b':2}
my_null = None
```

Syntax: Strings

```
print "hello world!"
print 'hello world!'
print '"hello world!"'
print '''
      hello
      world!
      '''
print 'hello' + ' ' + 'world!'
print '%s %s' % ('hello', 'world')
```

Syntax: Comments

```
# This is a single line comment
x = 5 # It can come after some code too

'''
This is a comment block.
Often used for documentation
   auto-documentation (Eg Sphinx)
'''
```

Syntax: Gotcha! Indentation

```
# In python, indentation matters; this wont work
print 'hello
        world!'

# There are some special exceptions, like comment blocks
print '''
        hello
        world!
    '''
```

- Both tabs and 4 whitespace characters are valid, but can’t be mixed in one script
- Best practice is 4 whitespaces, configure in IDE

Syntax: Lists

```
# First make a list:
L1 = [1, 2, 3]

# Adding additional elements:
L1.append(4)     # result: [1,2,3,4]

# Operator overloading:
L2 = ['a', 'b']
L3 = L1+L2       # result: [1,2,3,4,'a','b']
print len(L3)    # result: 6
print L3[2:4]    # result: [3, 4]
print L3[4:]     # result: ['a', ‘b']
print L3[-1]     # result: [‘b']

# List comprehension:
zero_through_4 = range(5) # result: [0, 1, 2, 3, 4]
even_list_leq_4 = [x for x in range(5) if x%2 == 0]

# Joining:
print ', '.join(['a','b','c'])
```

Syntax: Tuples

```
# In contrast to lists Tuples are not "mutable" (changeable)
t1 = (1,2,3)
# t1[0] = 7    # This will fail:

# Defining a tuple with one element:
t2 = (1,)

# An extra comma never hurts:
t3 = (1,2,3,)

# Tuples can be concatenated and sliced to form new tuples:
t1 = (1,2)
t2 = (3,4)
t3 = t1+t2
print t3    # (1,2,3,4)
t4 = t3[0:2]
print t4    # (1,2)
```

Mutable vs. Immutable

- Mutable: changeable (think “Mutationable”)
- Immutable: not changeable
- List are mutable, tuples are not

Wait, why does this work?

```
my_tuple = (1,2,3)
my_tuple = ('a','b')
```

- Mutable types can be changed in place.
- Immutable types can not change in place.
  The old is destroyed

Syntax: Dictionaries

```
# Create all at once:
D1 = {'a':1,'b':2}

# Create piece by piece:
D2 = {} # i.e. an empty dictionary
D2['a'] = 1
D2['b'] = 2

# More than one key value:
D2['s', 5] = 3

# Any immutable (including a tuple) can act as a key:
D2[(4,5)] = 4 # Equivalently D2[4,5] = 4

# Dictionaries have several useful methods:
print D2.keys()
print D2.values()
print D2.items()
print D2.get('c','default') # 2nd arg is a default

# Lists are mutable, so cannot be keys:
D2[[4,5]] = 5   # results in error
```

Syntax: Loops

```
# For loops move across any iterable:
# All control flow statements use a ":", and require indentation
my_list = [1,2,3]
for x in my_list:
    print x

# Can handle multiple values:
my_dict = {}
my_list = [('a',1), ('b',2)]
for letter, number in my_list:
    my_dict[letter] = number

# A good use of the "iteritems" method:
for letter, number in my_dict.iteritems():
    print letter, number

# Simple time-stepper with a while loop:
t = 0
while t < 10:
    t += 1
```

Syntax: Conditionals

```
# All control flow statements use a ":", and require indentation
x1 = 3
x2 = 4
if x1 < x2:
    print "less"
elif x1 > x2:
    print "greater"
else:
    print "equal"  # No "end"

# Compound conditionals:
x3 = False
x4 = True
if x3 or (not x4):
    "Inconceivable!"
```

Overview

- Background: What is Python?
- Syntax (lots of code examples)
- Modules? Packages?? Classes??? Oh my…
- Scientific programming in Python
- “Gotcha”s, i.e. what NOT to do
- Tips and Tricks, i.e. what TO DO

Code Organization

- Ways to organize code in python:
  - Command line interface (i.e. not organized)
  - Script file (.py by convention), interpreted by python directly
  - Functions: Easy and fast; make small and use often
  - Module: like a script but “imported” by another file
  - Packages: Hierarchy of modules organized by folders
- Special mention: Classes
  - Package data with the code that operates on it (Encapsulation)
  - Reuse of implementation (Inheritance/Polymorphism)

Scripts

Scripts in python are just like matlab:

- A File that can be executed
- End in a ".py" extension (though not necessary)
- Called from the command line using the "python" executable:

```
[nicholasc@firefly]~/$ python distance_between_points.py
```

```
# Define two points as a tuple of coordinates:
p1 = (1, 2)
p2 = (3, 4)

print ((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)**.5
```

Functions:

- Functions are very easy to define, use them!

```
# Function that returns distance between two points:
def distance_between_points_2D(tuple_1, tuple_2):

    delta_0 = tuple_1[0] - tuple_2[0]
    delta_1 = tuple_1[1] - tuple_2[1]

    return (delta_0**2 + delta_1**2)**.5

# Define two points as a tuple of coordinates:
p1 = (1, 2)
p2 = (3, 4)

print distance_between_points_2D(p1, p2)
```

Functions:

- Functions can have default arguments (keyword args)

```
def norm(x, p=2):
    return (abs(x[0])**p + abs(x[1])**p)**(1./p)

print norm((1,1))
print norm((1,1), p=1)
```

- Aside: Anonymous functions (lamda) are fast:

```
print map(lambda x:x**2, [1,2,3])
# [1, 4, 9]
```

Modules

A module enables code reuse, by defining “stuff” outside the script and “importing”. Requires the module is in the “python path”

```
# Define the points:
p1 = (1, 2)
p2 = (3, 4)

# Method 1: Import the function you need by name:
from distance_tools import distance_between_points_2D
print distance_between_points_2D(p1, p2)

# Method 2: Import the function you need and rename:
from distance_tools import distance_between_points_2D as d
print d(p1, p2)

# Method 3: Import the whole module, and use an attribute:
import distance_tools
print distance_tools.distance_between_points_2D(p1, p2)

# Method 4: Import the whole module, rename, and use an attribute:
import distance_tools as dt
print dt.distance_between_points_2D(p1, p2)
```

Packages

- Packages help structure large reusable code bases
- Require a file, “__init__.py” at each level of the folder hierarchy
- Top-level directory must be in the python path

```
from organization.package_example.tools import distance_tools as dt
from organization.package_example.tools import utilities as util
import time

# Define the points:
p1 = (1, 2)
p2 = (3, 4)

t0 = time.time()
for ii in range(1000000):
    curr_distance = dt.distance_between_points_2D(p1, p2)
print util.seconds_to_str(time.time()-t0)
```

Standard Packages

- Python comes with many default packages:
  - sys
  - os (os.path)
  - random
  - argparse
  - copy
  - pickle
  - io
  - time, time it, date time
  - … see https://docs.python.org/2/library/ for more

```
# If this file was called foo.py,
# Call "python foo.py 1 2"
import sys
print sys.argv
# [foo.py, "1", "2"]
```

External Packages

- Most external packages can be downloaded from PyPI:
  - https://pypi.python.org/pypi

```
pip install package_name
```

- If you don't have the pip executable in your command path, you can get it from: http://pip.readthedocs.org/en/latest/installing.html

- Other 3rd party packages should contain a "setup.py" file.
- Typical installation is simply "python setup.py install"
- Typical install location is the "site-packages" subdirectory

Getting the python voxel wrapper code:

- Installable python module on github:
  - https://github.com/AllenBrainAtlas/friday-harbor
- This code can manipulate data from the data directories (more on this tomorrow!)
- You can practice your python and look through the examples directory tonight.

Classes

- Classes are the fundamental idea of object-oriented programming (OOP)
- Allow us to bundle code and data (encapsulation), and lots of other fancy stuff.
- Simple in python (no, really)

```
class Vector(object):

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __sub__(self, other_point):
        del_x = self.x - other_point.x
        del_y = self.y - other_point.y
        return Vector(del_x, del_y)

    def norm(self):
        return (self.x**2 + self.y**2)**.5

    def __str__(self):
        return "(%s, %s)" % (self.x, self.y)
```

```
from vector import Vector as Point

# Define the points:
p1 = Point(1, 2)
p2 = Point(3, 4)

print (p1-p2).norm()
```

Inheritance:

```
import pylab as pl
import numpy.random as npr
import numpy as np

class Neuron(object):

    def __init__(self, v=0,
                       v_reset=0,
                       v_threshold=10):

        self.v = v
        self.v_threshold = v_threshold
        self.v_reset = v_reset

    def propogate(self, dt):
        self.v += np.sqrt(dt)*npr.randn()
        if self.v >= self.v_threshold:
            self.v = self.v_reset
```

```
import numpy as np
import numpy.random as npr
from neuron import Neuron

class PrintingNeuron(Neuron):

    def __init__(self, **kwargs):
        # Super returns a superclass object
        super(PrintingNeuron, self).__init__(**kwargs)

# Overriding the parent method:
def propogate(self, dt):
    self.v += np.sqrt(dt)*npr.randn()
    if self.v >= self.v_threshold:
        self.v = self.v_reset
        print 'I spiked!'
```

Overview

- Background: What is Python?
- Syntax (lots of code examples)
- Modules? Packages?? Classes??? Oh my…
- Scientific programming in Python
- “Gotcha”s, i.e. what NOT to do
- Tips and Tricks, i.e. what TO DO

Numpy, Scipy, and Pylab:

- Much of the matlab-like functionality of python is supported by numpy/scipy/pylab
- Numpy for MATLAB users:

- http://mathesaurus.sourceforge.net/matlab-numpy.html

```
import numpy as np
import pylab as pl

a = np.linspace(0, 2*np.pi, 1000)
pl.plot(a, np.sin(a))
pl.show()
```

Numpy, Scipy, and Pylab:

- Numpy is a little more particular about array shape
- Indexing is off by one

```
import numpy as np

A = np.array([[1,2],[3,4]])
B = np.array([[1],[0]])
print np.dot(A,B)
"""
[[1]
 [3]]
"""


print A[1,1] # 4
```

Numpy, Scipy, and Pylab:

- But you can iterate through the matrix:

```
import numpy as np

A = np.array([[1,2],[3,4]])

for row in A:
    for val in row:
        print val

'''
1
2
3
4
'''
```

Numpy, Scipy, and Pylab:

- Conditional indexing:

```
import numpy as np

A = np.array([[1,2],[3,4]])

print A[A>2] # [3 4]
```

- Like matlab find:

```
import numpy as np

A = np.array([[1,2],[3,4]])

inds = np.where(A>2)
   # (array([1, 1]), array([0, 1]))
print A[inds]
```

Numpy, Scipy, and Pylab:

- Read and write matlab format:

```
import numpy as np
import scipy.io as sio

A = np.array([[1,2],[3,4]])
sio.savemat('my_file.mat', {'A':A})
D = sio.loadmat('my_file.mat')
A = D['A']
print A
'''
[[1 2]
 [3 4]]
'''
```

Scikits and Pandas

- Short for Scipy-toolkits
- Specialized, and under development
- Examples:
  - learn: Machine learning
  - image: Image processing
  - statsmodels: desciptive statistics/model fitting

- Pandas: Large data set manipulation/analysis
  - Kinda like R for python
  - Compliments statsmodels and learn

Scikits and Pandas

- Talk to Shawn for more information on Pandas and ipython notebooks:

Connectivity experiment metadata as pandas DataFrame

shawno@allensinstitute.org

Imports and config for notebook display

In [1]:
```
import pandas as pd
pd.set_option('display.notebook_repr_html',True)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 5000)
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline
```

Get data

In [2]:
```
from friday_harbor.data.regenerate_data import regenerate_data
data_dir = "/Users/Shawn/Desktop/friday_harbor_data"
# regenerate_data(data_dir)
```

Instantiate ExperimentManager object

In [3]:
```
from friday_harbor import experiment
data_dir = "/Users/Shawn/Desktop/friday_harbor_data"
em = experiment.ExperimentManager(data_dir)
```

In [4]:
```
print "Number of experiments = ", len(em.experiment_list)
```

```
Number of experiments =  1772
```

ALLEN INSTITUTE *for* BRAIN SCIENCE
Fueling Discovery

A note on speed:

- Like MATLAB, code can either be slow or fast depending on your implementation
  - Optimize the bottleneck
  - Stay away from loops
  - Use map(), filter() or reduce(), with a built-in function instead
- Great case study:
  - https://www.python.org/doc/essays/list2str/

Overview

- Background: What is Python?
- Syntax (lots of code examples)
- Modules? Packages?? Classes??? Oh my…
- Scientific programming in Python
- “Gotcha”s, i.e. what NOT to do
- Tips and Tricks, i.e. what TO DO

Gotcha: Indenting

- Most new python coders mess up indenting.
- When should you indent?
  - Loops (for, while)
  - Conditionals (if/elif/else statements)
  - Function definitions
  - Class definitions

Python forces you to write clean code. For other style guidelines, see:

http://legacy.python.org/dev/peps/pep-0008/

Gotcha: Integer division

- Yeah…

```
print 5/2       # 2
print 5./2      # 2.5
```

Gotcha: References vs. Values

- Mutables are like references, this can trip you up:

A is mutable

```
A = [1,2,3]
B = A
A[0] = 0
print B
```

Prints: [0, 2, 3]

A is immutable

```
A = 1
B = A
A = 0
print B
```

Prints: 1

Gotcha: is vs. ==

- In python, == is not the same as “is”
- == tests *value equality*, is tests *reference equality*

```
A = [1]
B = [1]
print A == B  # True
print A is B  # False
```

- For a general object, == behavior can be defined with the __eq__ method

Gotcha: Mutable Default Args

- Default function arguments are awesome, but can get you in trouble if they are mutable type:

```
def append_to(element, to=[]):
    to.append(element)
    return to

my_list = append_to(12)
print my_list                # [12]

my_other_list = append_to(42)
print my_other_list          # [12, 42]
```

Overview

- Background: What is Python?
- Syntax (lots of code examples)
- Modules? Packages?? Classes??? Oh my…
- Scientific programming in Python
- “Gotcha”s, i.e. what NOT to do
- Tips and Tricks, i.e. what TO DO

Enumerate:

- The enumerate function iterates with an index:

```
import numpy as np
x = np.linspace(10,15,6)
y = np.empty_like(x)
for ii, x in enumerate(x):
    y[ii] = x**2
```

Sorting:

- Sorting is easy and fast, and can be done with keys
- “sort” method is in-place, “sorted” returns a new list

```
x = [(5, 'a'),(8,'b'),(3,'c'),(4,'d')]
print sorted(x, key=lambda x:x[0])
print x
print x.sort(key=lambda x:x[0])
print x
```

```
[(3, 'c'), (4, 'd'), (5, 'a'), (8, 'b')]
[(5, 'a'), (8, 'b'), (3, 'c'), (4, 'd')]
None
[(3, 'c'), (4, 'd'), (5, 'a'), (8, 'b')]
```

Assert:

- If something is impossible in your code, use assert to find bugs faster
- Allows very fast unit testing:

```
# Function that returns distance between two points:
def distance_between_points_2D(tuple_1, tuple_2):
    delta_0 = tuple_1[0] - tuple_2[0]
    delta_1 = tuple_1[1] - tuple_2[1]

    return (delta_0**2 + delta_1**2)**.5

if __name__ == "__main__":
    assert distance_between_points_2D((1,1),(4,5)) == 5

    from numpy.random import rand
    p1 = rand(2)
    assert distance_between_points_2D(p1, p1) >= 0
```

Functions:

- Functions are objects, so they can be thrown around like anything else

```
# Function as argument:
def iterate_map(x0, f, N):
    curr_x = x0
    for ii in range(N):
        curr_x = f(curr_x)
    return curr_x

def f(x):
    return 4*x*(1-x)

print iterate_map(.1,f, 2)
```

```
# Function as return value
def function_factory(exp):
    def f(x):
        return x**exp
    return f

f = function_factory(3)
print f(3)
```

__name__ == “__main__”:

- A common way to prevent code from being executed when importing is with the following construct:

```
# Function that returns distance between two points:
def distance_between_points_2D(tuple_1, tuple_2):
    delta_0 = tuple_1[0] - tuple_2[0]
    delta_1 = tuple_1[1] - tuple_2[1]

    return (delta_0**2 + delta_1**2)**.5

if __name__ == "__main__":

    print "this code wont be executed on import"
    print "but will if run as a script"
```

args and kwargs:

- Function signatures can be very flexible

```
def f(*args, **kwargs):
    print args, kwargs

f(1, 2, c='a', d=7)                    (1, 2) {'c': 'a', 'd': 7}

arg_tuple = (1,2)
f(*arg_tuple)                          (1, 2) {}

arg_dict = {'c':'a', 'd':7}
f(**arg_dict)                          () {'c': 'a', 'd': 7}
```

Memoizing function:

- That last one can actually be “exploited” as a feature
- Here is a recursive, memoizing, fibonacci number function

```
def fibonacci(n, d={0:1,1:1}):
    try:
        return d[n]
    except:
        if n < 0:
            raise Exception
        d[n] = fibonacci(n-1) + fibonacci(n-2)
        return d[n]
```

Properties:

- Attributes with computation, always fresh:

```
import pylab as pl
import numpy.random as npr
import numpy as np

class Neuron(object):

    def __init__(self, v=0, v_reset=0, v_threshold=5):
        assert v < v_threshold

        self.v = v
        self.v_threshold = v_threshold
        self.v_reset = v_reset

    def propogate(self, dt):
        self.v += np.sqrt(dt)*npr.randn()
        if self.v >= self.v_threshold:
            self.v = self.v_reset

    @property
    def distance_to_threshold(self):
        return self.v_threshold - self.v

n1 = Neuron()
print n1.distance_to_threshold
```

Dynamic Modification:

- Use like salt: only a pinch, to taste

```
import pylab as pl
import numpy.random as npr
import numpy as np

class Neuron(object):

    def __init__(self, v=0,
                       v_reset=0,
                       v_threshold=10):

        assert v < v_threshold

        self.v = v
        self.v_threshold = v_threshold
        self.v_reset = v_reset

    def propogate(self, dt):
        self.v += np.sqrt(dt)*npr.randn()
        if self.v >= self.v_threshold:
            self.v = self.v_reset
```

```
# Settings:
t0 = 0
dt = .001
tf = 5
v_threshold = 1

# Initializations:
neuron_1 = Neuron(v_threshold=v_threshold)

# Patch on a new propogate function:
old_propogate = neuron_1.propogate
def propogate(dt):
    old_propogate(dt)
    if neuron_1.v_reset == neuron_1.v:
        print 'spike!'
neuron_1.propogate = propogate

# Time Loop
t = t0
while t < tf:
    t += dt
    neuron_1.propogate(dt)
```