

# Python Quick Start for Economists

Chicago Federal Reserve Workshop

John Stachurski

May 2016

# This Morning

- 9:00–10:30 Lecture
- 10:30–11:00 Coffee break
- 11:00–12:30 Exercises

## Lecture topics:

- Introduction
- A first program
- Some language features
- Scientific programming

# Set Up

## Use Anaconda!

- Bundles Python and the main scientific libraries
- Free from <http://continuum.io/downloads>
- Make it your default Python distribution

## Keeping up to date:

- In a terminal type `conda update anaconda`
  - On Windows, terminal = `cmd`

# Downloads

Documents for today's lecture available from

- [https://github.com/QuantEcon/ChicagoFed\\_workshop](https://github.com/QuantEcon/ChicagoFed_workshop)

Download the whole repo using Download ZIP (or git)

In `Tuesday_morning` you'll find

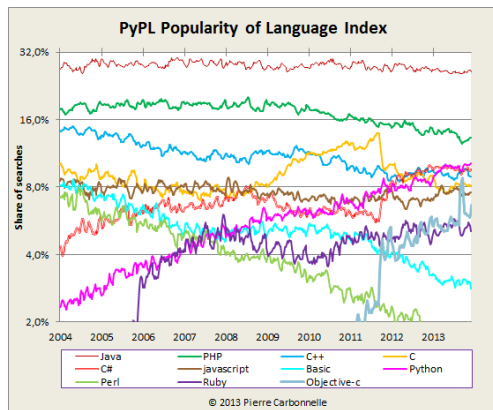
- these lecture slides (PDF)
- `scientific_python_quickstart.ipynb`
- `python_exercises.ipynb`

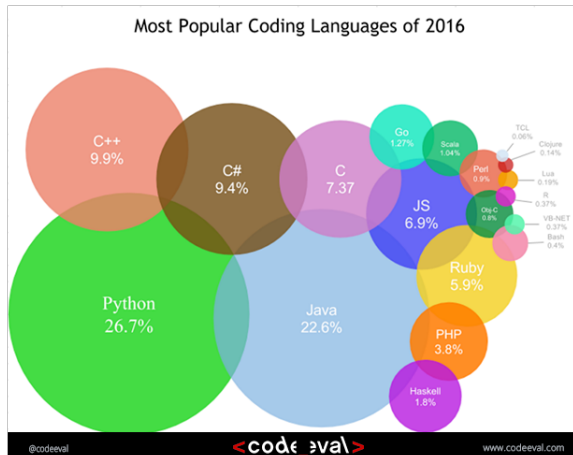
# What's Python?

A high level, general purpose programming language

Used extensively by

- Tech firms (YouTube, Dropbox, Reddit, etc., etc.)
- Hedge funds and finance industry
- Gov't agencies (NASA, CERN, etc.)
- Academia





Noted for

- Elegant, modern design
- Clean syntax, readability
- High productivity

Often used to teach first courses in comp sci and programming

- MIT, Stanford, Chicago, NYU, Yale, etc.
- Udacity, edX, etc.

Very popular in “data science” / machine learning



# Why I Like Python

1. Well designed language
  - Elegant, readable, expressive
2. General purpose
  - meets all my coding needs
  - large set of mature libraries
3. Other awesomeness
  - Numba, Jupyter, etc
4. Open source
  - no license hassles
  - can read / edit source code

# Interacting with Python

How do we write / run code in Python?

Options

- IDE
- Text editor plus REPL
- Jupyter notebooks
- etc.

# Spyder

Spyder (Python 3.5)

File Edit Search Source Run Debug Consoles Tools View Help

Editor: /home/john/sync\_dir/teaching/nyu/more\_python/plot\_example\_5.py

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy.stats import norm
4 from random import uniform
5 num_rows, num_cols = 2, 3
6 fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 8))
7 for i in range(num_rows):
8     for j in range(num_cols):
9         m, s = uniform(-1, 1), uniform(1, 2)
10        x = norm.rvs(loc=m, scale=s, size=100)
11        axes[i, j].hist(x, alpha=0.6, bins=20)
12        t = r'$\mu = {0:.1f}, \sigma = {1:.1f}'.format(m, s)
13        axes[i, j].set_title(t)
14        axes[i, j].set_xticks([-4, 0, 4])
15        axes[i, j].set_yticks([])
16 plt.show()
17
18

```

Variable explorer

Name	Type	Size	Value
axes	object	(2, 3)	array([[<matplotlib.axes._su...
i	int	1	1
j	int	1	2
m	float	1	0.2753487585185199
num_cols	int	1	3
num_rows	int	1	2
s	float	1	1.177823269914244
t	str	1	'\$\mu = 0.3, \sigma = 1.2\$'

Variable explorer File explorer

IPython console

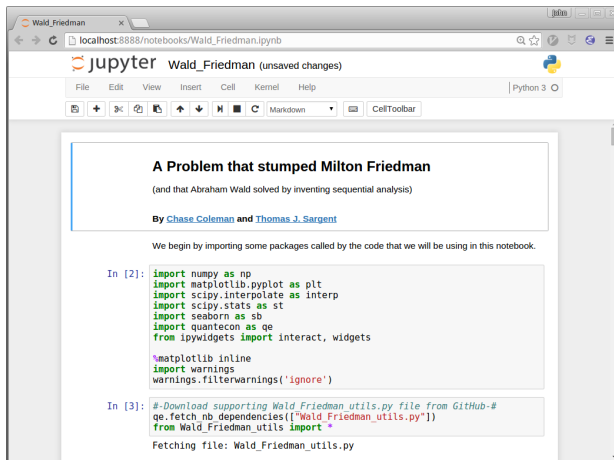
Console 1/A

In [2]:

Console History log IPython console

Permissions: RW End-of-lines: LF Encoding: UTF-8-GSS548 Line: 3 Column: 3 Memory: 23 %

# Jupyter notebooks



# My set up (Linux terminal, Tmux, Vim, IPython REPL)

```

51     x0 = x1
52
53
54     return x1
55
56 @jit(nopython=True)
57 def updated_function(n1, n2):
58     hn1 = h(n1)
59     hn2 = h(n2)
60     if n1 <= 0.5 and n2 <= 0.5:
61         new_n1 = delta * (0.5 * theta + (1 - theta) * n1)
62         new_n2 = delta * (0.5 * theta + (1 - theta) * n2)
63     elif n1 >= hn2 and n2 >= hn1:
64         new_n1 = delta * n1
65         new_n2 = delta * n2
66     elif n1 > 0.5 and n2 <= hn1:
67         new_n1 = delta * n1
68         new_n2 = delta * (theta * hn1 + (1 - theta) * n2)
69     else:
70         new_n1 = delta * (theta * hn2 + (1 - theta) * n1)
71         new_n2 = delta * n2
72     return new_n1, new_n2
73
74 def plot_trajectory(n1, n2, k):
75     """
76     Plot trajectory of length k from (n1, n2).
77     """
78     fig, ax = plt.subplots()
79     traj = np.empty(k)
80     for i in range(k):
81         new_n1, new_n2 = updated_function(n1, n2)
82         n1, n2 = new_n1, new_n2
83         traj[i] = abs(n1 - n2)
84     ax.plot(traj, 'b-')
NORMAL  ./matsuyama_synchronization.py  < python  72%  84.5
12  0x python3

```

```

( 2 ) ))) pwd                                     +1265 10:15 (-)
/home/john/sync_dir/teaching/nyu/need_for_speed

( 2 ) ))) ipython                                +1266 10:15 (-)
Python 3.5.1 [Anaconda 2.4.1 (64-bit)] (default, Dec  7 2015, 11:16:01)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.1 -- An enhanced Interactive Python.
?                -> Introduction and overview of IPython's features.
%quickref        -> Quick reference.
help             -> Python's own help system.
object?         -> Details about 'object', use 'object??' for extra details.

In [1]:

```

11d 4m 15s 2.2 2.5 2.1 2016-01-22 10:16 godzilla-nyc

# Jupyter Notebooks

Today we'll interact with Python using Jupyter notebooks

- A browser based front end to Python, Julia, R, etc.
- Stores output as well as input
- Allows for rich text, graphics, etc.
- Easy to run remotely on servers / in cloud

## To show

- Editing modes, execution
- Markdown
- Inline figures
- Language agnostic
- Ref: [http://quant-econ.net/py/getting\\_started.html](http://quant-econ.net/py/getting_started.html)
- Examples: <http://notebooks.quantecon.org/>

# An Easy Python Program

Next step: write and pick apart small Python program

Aim: To simulate and plot

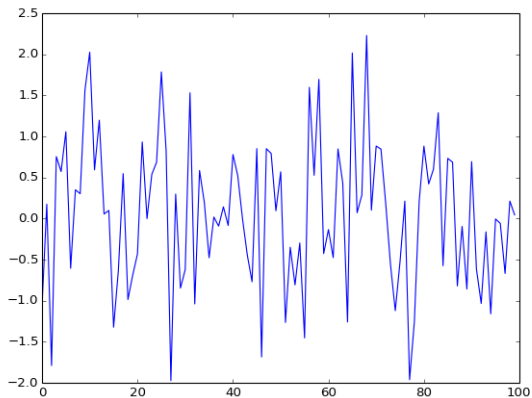
$$\epsilon_0, \epsilon_1, \dots, \epsilon_T \quad \text{where} \quad \{\epsilon_t\} \stackrel{\text{iid}}{\sim} N(0,1)$$

Notes

1. Like all first programs, to some extent contrived
2. We focus as much as possible on pure Python



## Desired output (modulo randomness)



## First pass

---

```
import matplotlib.pyplot as plt
from random import normalvariate

ts_length = 100
epsilon_values = []
for i in range(ts_length):
    e = normalvariate(0, 1)
    epsilon_values.append(e)
plt.plot(epsilon_values, 'b-')
plt.show()
```

---

- [http://quant-econ.net/py/python\\_by\\_example.html](http://quant-econ.net/py/python_by_example.html)

# Import Statements

Consider the lines

---

```
1 import matplotlib.pyplot as plt
2 from random import normalvariate
```

---

We are importing functionality from two modules

- module = file containing Python code

Importing a module causes Python to

- run the code in those files
- set up a matching "namespace" to store variables

```
In [1]: import random
```

```
In [2]: random.normalvariate(0, 1)
```

```
Out[2]: 0.18415513098683509
```

```
In [3]: random.uniform(0, 1)
```

```
Out[3]: 0.11883707116624409
```

You can also just import the names directly:

```
In [4]: from random import normalvariate, uniform
```

```
In [5]: normalvariate(0, 1)
```

```
Out[5]: -0.7248742909651001
```

```
In [6]: uniform(0, 4)
```

```
Out[6]: 0.24696251658189228
```

# Lists

Statement `epsilon_values = []` creates an empty list

Lists: a Python data structure used to group objects

```
In [7]: x = [10, 'foo']
```

```
In [8]: type(x)
```

```
Out[8]: list
```

Note that different types of objects can be combined in a single list

Adding a value to a list: `list_name.append(some_value)`

```
In [10]: x
```

```
Out[10]: [10, 'foo']
```

```
In [11]: x.append(0.5)
```

```
In [12]: x
```

```
Out[12]: [10, 'foo', 0.5]
```

Here `append()` is an example of a **method**

- a function “attached to” an object

Another example of a list method:

```
In [13]: x
```

```
Out[13]: [10, 'foo', 0.5]
```

```
In [14]: x.pop()
```

```
Out[14]: 0.5
```

```
In [15]: x
```

```
Out[15]: [10, 'foo']
```

To see all list methods, type `x.` and hit “Tab”



Lists in Python are **zero based**

- Like C, Java, Ruby, Go, etc.

```
In [16]: x
```

```
Out[16]: [10, 'foo']
```

```
In [17]: x[0]
```

```
Out[17]: 10
```

```
In [18]: x[1]
```

```
Out[18]: 'foo'
```

# Looping

Consider again these lines from `test_program_1.py`

---

```
6 for i in range(ts_length):  
7     e = normalvariate(0, 1)  
8     epsilon_values.append(e)  
9 plt.plot(epsilon_values, 'b-')
```

---

Lines 7–8 are the **code block** of the `for` loop

Reduced indentation signals end of code block

## Comments on Indentation

In Python **all** code blocks are delimited by indentation

Pros: more consistency, less clutter

Cons: a bit fragile

Notes:

- Line before start of code block always ends in a colon
- All lines in a code block must have same indentation
- The Python standard is 4 spaces

# Language Features

Now let's turn to some features of the Python language

- Common data types
- Objects and methods
- Name and variables

# Data Types

Some native Python data types

```
In [1]: s = 'foo'
```

```
In [2]: type(s)
```

```
Out[2]: str
```

```
In [3]: x = 0.1
```

```
In [4]: type(x)
```

```
Out[4]: float
```

```
In [5]: y = ['foo', 'bar']
```

```
In [6]: type(y)
```

```
Out[6]: list
```

## Type is important in Python

```
In [1]: 1 + 1
```

```
Out[1]: 2
```

```
In [2]: 'foo' + 'bar'
```

```
Out[2]: 'foobar'
```

```
In [3]: ['foo', 'bar'] + [10, 20]
```

```
Out[3]: ['foo', 'bar', 10, 20]
```

Each data type has its own methods

```
In [7]: y = ['foo', 'bar']
```

```
In [8]: y.reverse()
```

```
In [9]: y
```

```
Out[9]: ['bar', 'foo']
```

An example of a string method:

```
In [10]: s = 'foobar'
```

```
In [11]: s.capitalize()
```

```
Out[11]: 'Foobar'
```

Another data type is a **tuple** — similar to a list

```
In [1]: x = ['a', 'b']    # Square brackets for lists
```

```
In [2]: type(x)
```

```
Out[2]: list
```

```
In [3]: x = ('a', 'b')   # Round brackets for tuples
```

```
In [4]: type(x)
```

```
Out[4]: tuple
```

```
In [5]: x = 'a', 'b'     # No brackets is identical
```

```
In [6]: type(x)
```

```
Out[6]: tuple
```



In fact tuples are **immutable** lists

**Immutable** means internal state cannot be altered

```
In [1]: x = (10, 20)
```

```
In [2]: x[0] = 'foo'
```

```
-----  
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-2-ff02f57fd8a0> in <module>()  
----> 1 x[0] = 'foo'
```

```
TypeError: 'tuple' object does not support item assignment
```

# Functions

Some are built-in:

```
In [1]: max(10, 20, -2)
```

```
Out[1]: 20
```

```
In [2]: max
```

```
Out[2]: <function max>
```

Others are imported:

```
In [3]: from math import sqrt
```

```
In [4]: sqrt
```

```
Out[4]: <function math.sqrt>
```

We can also write our own functions

```
In [1]: def f(x):  
...:     return x + 42  
...:
```

```
In [2]: f(1)
```

```
Out[2]: 43
```

One line functions using the `lambda` keyword:

```
In [3]: f = lambda x: x + 42
```

# Object Oriented Programming

Traditional programming paradigm is called **procedural**

- A program has state (values of its variables)
- Functions are called to act on this state
- Data is passed around via function calls

In **OOP**, data and functions bundled together into **objects**

These bundled functions are called **methods**

Example: Lists = list data + list methods

```
In [1]: x = []
```

```
In [2]: x.append('foo')
```

```
In [3]: x
```

```
Out[3]: ['foo']
```

Compare with Julia:

---

```
julia> x = Any[]
```

```
0-element Array{Any,1}
```

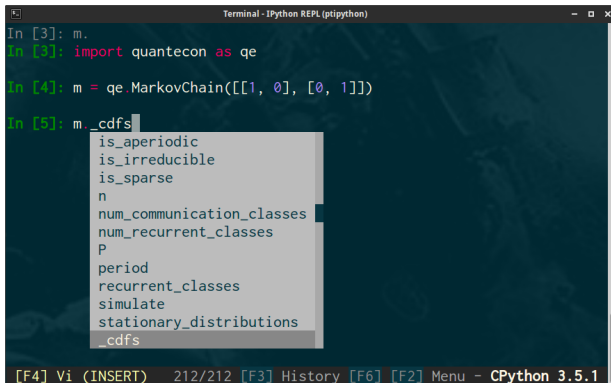
```
julia> push!(x, "foo")
```

```
1-element Array{Any,1}:
```

```
"foo"
```

---

## One advantage: introspection



```
Terminal - IPython REPL (ptipython)

In [3]: m.
In [3]: import quantecon as qe

In [4]: m = qe.MarkovChain([[1, 0], [0, 1]])

In [5]: m._cdfs
is_aperiodic
is_irreducible
is_sparse
n
num_communication_classes
num_recurrent_classes
P
period
recurrent_classes
simulate
stationary_distributions
_cdfs

[F4] Vi (INSERT) 212/212 [F3] History [F6] [F2] Menu - CPython 3.5.1
```

# Names and Objects

Consider this assignment statement

```
x = 42
```

We are **binding** the **name** `x` to the object on the right hand side

Thus, **names are symbols bound to objects stored in memory**

Python is **dynamically typed** — names are not specific to type

```
s = 'foo'    # Bind s to a string  
s = 100      # and now to an integer
```

Consider this (frightening?) code example

```
In [1]: x = ['foo', 'bar']
```

```
In [2]: y = x
```

```
In [3]: y[0] = 'fee'
```

```
In [4]: x
```

```
Out[4]: ['fee', 'bar']
```

Works because

- x and y are bound to the **same object**
- that object is **mutable**



# Scientific Programming with Python

Rapid adoption by the scientific community

- engineering
- computational biology
- chemistry
- physics, etc., etc.

More recently

- AI, machine learning, “data science”

# Key Scientific Libraries

## NumPy

- basic data types
- simple array processing operations

## SciPy

- built on top of NumPy
- provides additional functionality

## Matplotlib

- 2D and 3D figures

# NumPy

NumPy Example: Mean and standard dev of an array

```
In [1]: import numpy as np
```

```
In [2]: a = np.random.randn(100)
```

```
In [3]: a.mean()
```

```
Out[3]: -0.091480787986957607
```

```
In [4]: a.std()
```

```
Out[4]: 1.093037615548889
```

## Previous Example Using NumPy

Let's redo our earlier example using NumPy

---

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

epsilon_values = np.random.randn(100)
plt.plot(epsilon_values, 'b-')
plt.show()
```

---

# SciPy

**SciPy** Example: Calculate

$$\int_{-2}^2 \phi(z) dz \quad \text{where} \quad \phi \sim N(0,1)$$

```
In [1]: from scipy.stats import norm
```

```
In [2]: from scipy.integrate import quad
```

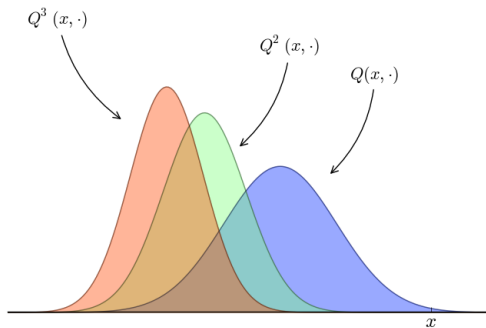
```
In [3]: phi = norm(0, 1)
```

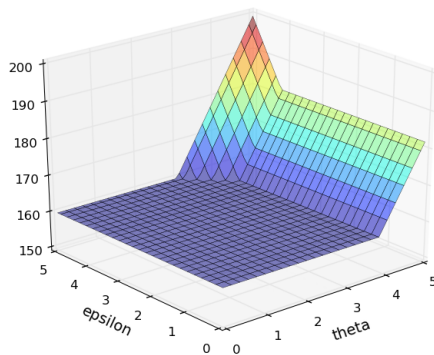
```
In [4]: value, error = quad(phi.pdf, -2, 2)
```

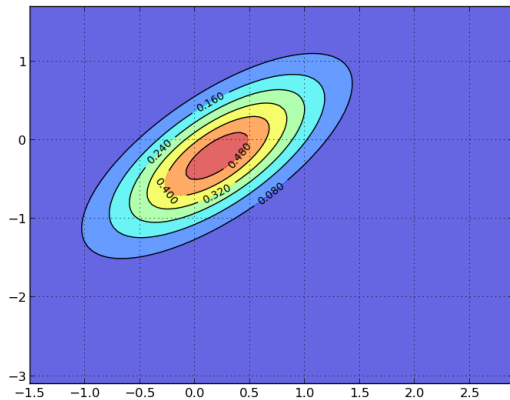
```
In [5]: value
```

```
Out[5]: 0.9544997361036417
```

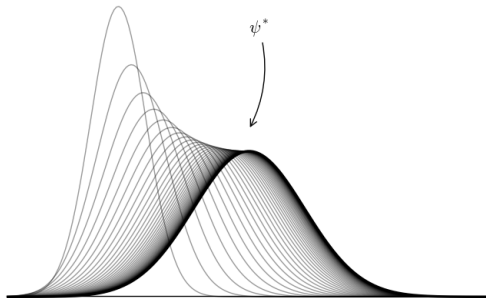
## Matplotlib examples











# Other Scientific Libraries

## Pandas

- statistics and data analysis

## SymPy

- symbolic manipulations à la Mathematica

Still more:

- **statsmodels** — statistics / econometrics
- **scikit-learn** — machine learning in Python

# Python Libraries for Econ

**QuantEcon** (<http://quantecon.org/>) provides

- Markov chains
- Dynamic programming
- LQ control
- etc

**Dolo** for quantitative macro

- A modeling language
- Many solution methods

# Other Scientific Tools

Also tools for

- working with graphs (as in networks)
- parallel processing, GPUs
- manipulating large data sets
- interfacing with C / C++ / Fortran
- cloud computing
- database interaction
- bindings to high level languages like R and Julia
- etc.

# Further Resources

Further discussion and links available at <http://quant-econ.net>

- Basic instructions
- Python lectures
- Scientific Python
- Lots of economic examples

## Exercise Session

Inside the `Tuesday_morning` dir of

- [https://github.com/QuantEcon/ChicagoFed\\_workshop](https://github.com/QuantEcon/ChicagoFed_workshop)

you will find

1. `scientific_python_quickstart.ipynb`
2. `python_exercises.ipynb`

Work through these notebooks in order

To do so,

1. start `jupyter-notebook`
2. click 'Upload'
3. navigate to file