Physics 403

Programming Bootcamp:
Data Analysis with Python

Segev BenZvi

Department of Physics and Astronomy
University of Rochester

Table of Contents

What is Python?

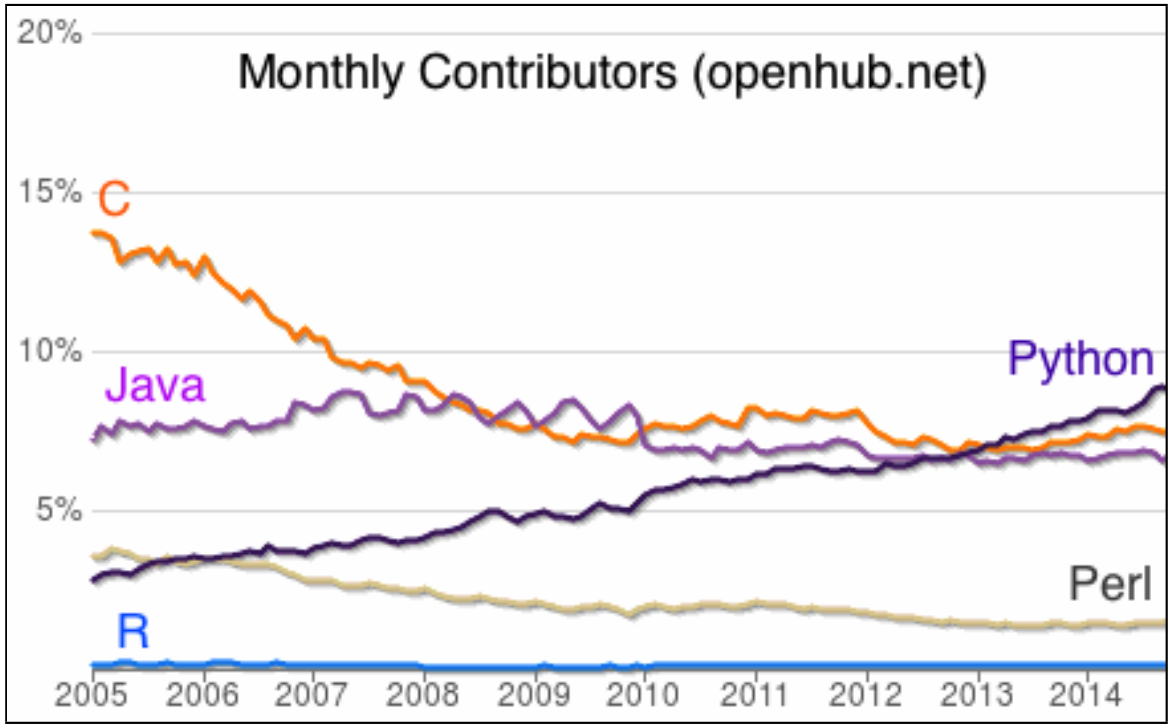
Python is an *imperative*, *interpreted* programming language with *strong dynamic* typing.

- Imperative: programs are built around one or more subroutines, known as "functions" and "classes."
- Interpreted: program instructions are executed directly rather than being compiled into machine code; compare to C, C++, ...
- Dynamic Typing: data types of variables (`int`, `float`, `string`, etc.), are determined on the fly as the program runs.
- Strong Typing: converting a value of one type to another (e.g., `string` to `int`) is not done automatically.

Python offers fast and flexible development, and you can copy programs between machines without worrying about compatibility. But it's slower than compiled Fortran, C, or C++ programs.

Why Use Python?

Python is one of the most popular scripting languages in the world, with a large community of users and hundreds of third-party modules [1, 2].



It is also a "glue language" with bindings to popular software used in physics and astronomy (for example, ROOT).

Key Third-Party Packages for Data Analysis

Must-haves. These come by default on some systems (Mac OS X):

- numpy: random numbers, arrays, transcendental functions, linear algebra.
- scipy: statistical tests, special functions, integration, curve fitting, minimization.
- matplotlib: plotting: xy plots, error bars, contour plots, histograms.

Worth using in your analysis:

- SciKits: data analysis add-ons to the scipy package.
- astroML: statistical methods and machine learning for astronomy.
- emcee: implementation of Markov Chain Monte Carlo.

Honorable mentions:

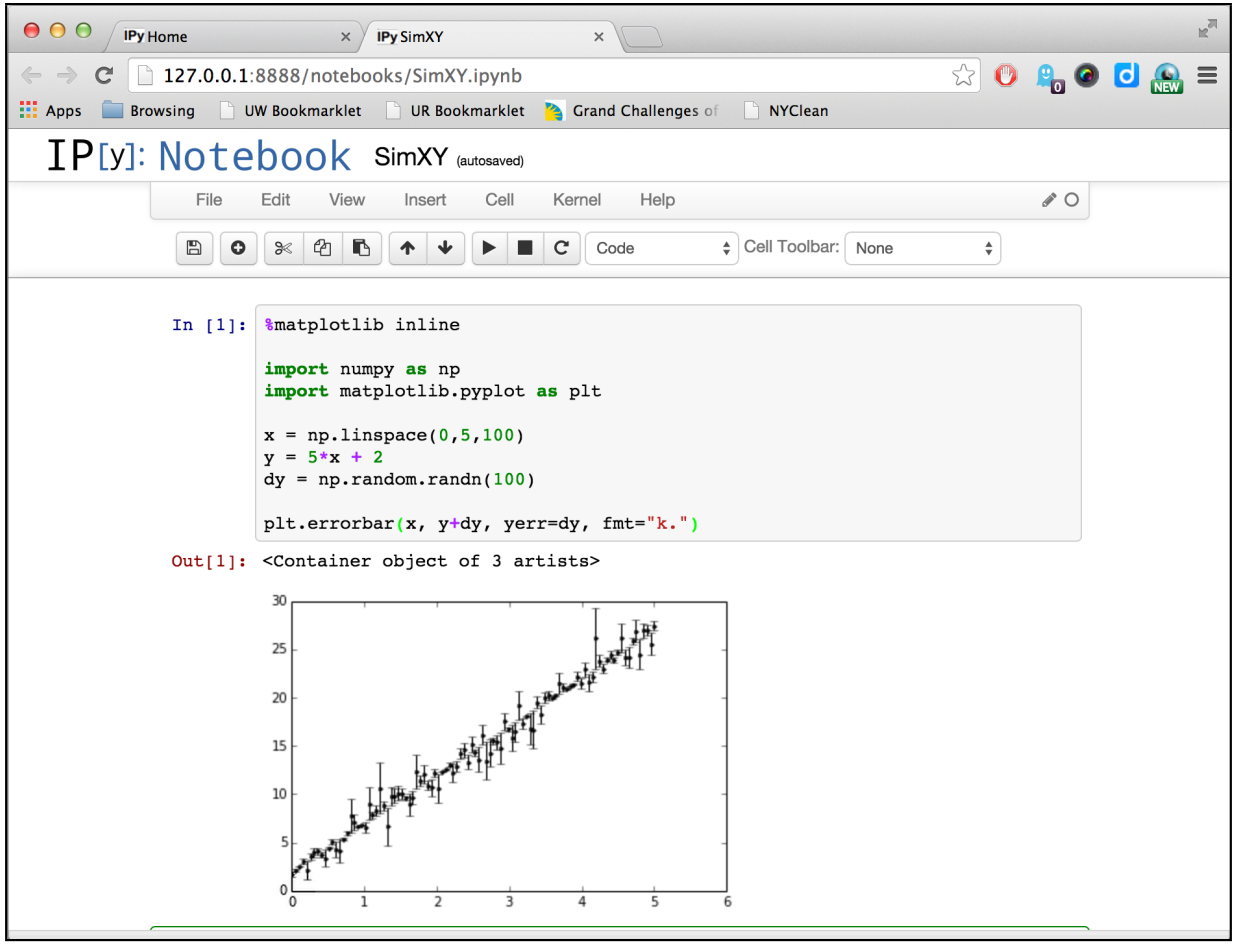
- lmfit, PyEphem and healpy (for astronomers), pandas.

Recommendation: most of these packages have image galleries with code examples. Going through them is the best way to learn.

Python Tools

IPython

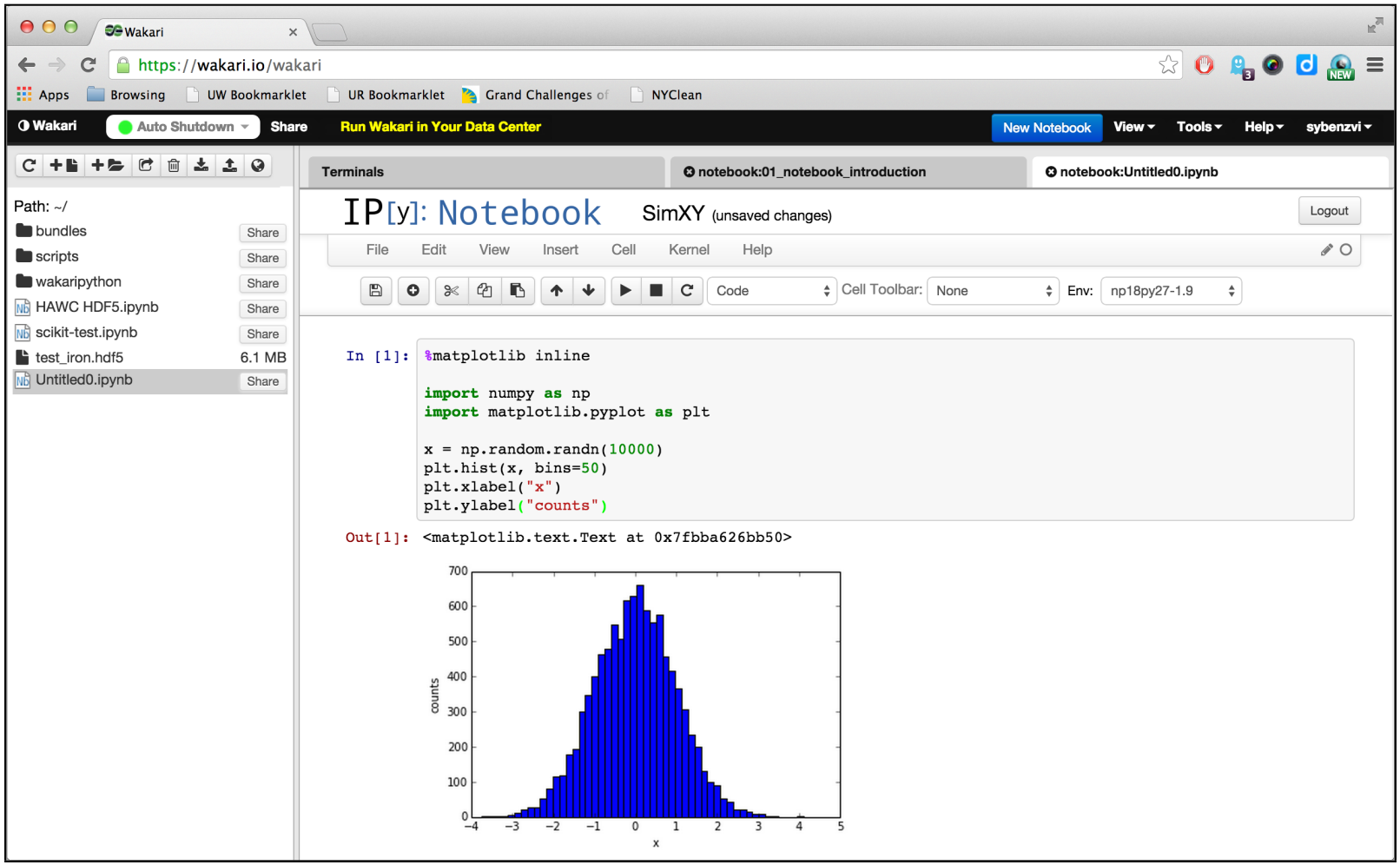
IPython [4] is highly recommended. Running `ipython notebook` in your terminal will open a Mathematica-style notebook where you can run code.



Python Tools

IPython: wakari.io

If you don't want to install IPython on your computer you can create a free account at wakari.io.



Installing Packages

Python has two package managers that makes installing third-party modules pretty simple:

- easy_install, an older, less user-friendly package manager.

Example

To install the pip program, in a terminal window type

```
sudo easy_install pip
```

- pip, a new package manager that supercedes easy_install.

Example

Install the scikit machine learning package in your home directory:

```
pip install --user scikits.learn
```

Table of Contents

A Basic Python Program

Formatted Printing

You can print messages, literal values, and the contents of variables using the `print` function:

```
print("Hello")
print(3.14159)
print("%s, %.2f" % ("Hello", 3.14159))
```

The resulting output:

```
Hello
3.14159
Hello, 3.14
```

The *format string* in the third `print` statement is explained in the Python documentation [6]. For those of you who know C, it's the same syntax as the `printf` function.

Variables and Arithmetic

Variables store a value that can be looked at or changed later.

```
x = 4                                  # x is an int
print("  x = %d" % x)                  # prints 4
print(" 2x = %d" % (2*x))              #    "   8
print("x^2 = %d" % (x**2))             #    "   16

y = 5.                                 # y is a float
print("  y = %g" % y)                  # prints 5
print("y/2 = %g" % (y/2))              #    "   2.5

z = "mystring"                         # z is a string
print("  z = %s" % z)

u = z + x                              # raise a TypeError
```

Arithmetic and Basic Operations

A word of caution about integer division in Python 2:

- $5/3 = 1$, not 1.667
- $1/2 = 0$, not 0.5
- I.e., integer division pre-Python 3 is *floor division*, only producing an *integer* quotient.
- Mistakenly assuming the output of integer division will give you a non-integer is a common bug, and often hard to track down.

If you use Python 3, this is no longer an issue. For everyone else, if you want the floating point value of the quotient of two integers you'll need to typecast the result:

$$\texttt{q} = \texttt{float(x)/y}.$$

Changing the Flow of Control

Conditional Statements

You can change the flow of control in a program using Boolean variables in an `if/elif/else` block. Note: block code is *indented*.

```
x = 5

if x > 10:                  # - evaluates to False;
    print("x > 10")         #    skip this...
elif x > 5:                 # - also False;
    print("x > 5")          #    skip this too...
else:                       # - block ends here,
    print("x <= 5")         #    so this is printed

isEven = (x % 2 == 0)       # store a boolean (False),
if not isEven:              # not False => True
    print "x is odd"        # so this is executed
```

Loops

The While Loop

While loops execute a block of code as long as a logical condition is satisfied. Note: loops (and conditional code) can be *nested*.

```
i = 0
while i < 10:          # Loop condition: i<10
    i += 1             # - increment i, break if i>=10
    if i % 2 == 0:
        print(i)       # Print i if it's even

print("All done.")
```

Once the loop condition is no longer satisfied the flow of control is returned to the main body of the program. Be careful about inadvertently writing an infinite loop, a serious runtime bug where the loop condition never evaluates to `False`.

Functions

User-Defined Functions

Functions are subroutines that accept some input and produce zero or more outputs. They are typically used to define common tasks in a program.

```
def round_int(x):                    # Round ints to the
    return 10 * ((x+5)/10)           # nearest 10

x = 2
while x < 1000:
    x *= 3                           # increase x in
    rx = round_int(x)                # multiples of 3
    print("%d -> %d" % (x, rx)) # until x > 1000
```

This program will calculate the list of numbers $\{6, 18, 54, 162, \ldots\}$ and round them to $\{10, 20, 50, 160, \ldots\}$.

In-Class Exercise

The Fibonacci Series

With the small amount you have learned, you can already implement reasonably sophisicated programs.

Example

The Fibonacci sequence $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ is defined by the linear homogeneous recurrence relation

$$F_n = F_{n-1} + F_{n-2},$$

where $n = 0, 1, 2, 3, \ldots$ and $F_0 = F_1 = 1$.

- Write a Python function that generates F_n given n.
- Write a program that generates all the Fibonacci numbers below 1000.

Let's take 5 minutes to work this out, then look at possible solutions...

In-Class Exercise

Recursive Fibonacci Function

This implementation is a recursive function, i.e., a function that calls itself. It's an easy way to implement the Fibonacci sequence.

```
def fib(n):
    """Generate term n of the Fibonacci series."""
    if n <= 1:
        # if n = 0 or 1: return 1
        return 1
    else:
        # else if n = 2, 3, ...: call fib
        return fib(n-1) + fib(n-2)
```

Unfortunately, the function *call stack* grows rapidly with n, so `fib` will slow down as n goes up. For sufficiently large n, you'll hit the Python call stack limit and the program will crash.

In-Class Exercise

Efficient Fibonacci Function

This more efficient implementation contains two internal *state variables* a and b, which keep track of the terms F_{n-1} and F_{n-2}.

```
def fib(n):
    """Generate the Fibonacci series."""
    a, b = 0, 1
    while n > 0:                    # loop condition
        a, b, n = b, a+b, n-1       # update step
    return b
```

We loop over n terms in the series by decrementing the function argument n and terminating the loop when $n = 0$.

In-Class Exercise

Fibonacci Function

There are many ways we could calculate all of the $\{F_n\}$ below 1000. A simple way is to use another `while` loop:

```
n = 0                   # Loop variable: n
f = fib(n)              # Loop condition: fib(n)<1000
while f < 1000:
    print("%4d%8d" % (n, f))
    n +=1
    f = fib(n)          # Update for next iteration
```

There are two variables that track the state of the loop: `n` and `f=fib(n)`. We break out of the loop when `f>=1000`.

The result: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987$.

Lists and Tuples

Lists offer the very convenient ability to store groups of related values in a single data structure.

```
m = [5,6,7,8,9]      # A list of integers
n = range(5,10)      # A list n=m made with range()

print(m[0])          # print element 0:    5
print(n[1])          # print element 1:    6
print(n[-1])         # print last element: 9
print(n)             # prints [5, 6, 7, 8, 9]
```

This program makes two equal lists of integers; the second one is produced using the built-in function `range`. Access to elements in the list is provided by the bracket operator []. Elements are indexed $0, 1, 2, \ldots$ from the front, or $-1, -2, -3, \ldots$ from the back.

Lists and Tuples

Tuples are the same as lists but cannot be modified once initialized. They are "read-only" data structures.

```
t1 = ["Jan", "Feb", "Mar"]        # a list
t2 = ("Apr", 10, 2020)            # a tuple

print(t1[0])                      # "Jan"
print(t2[-2])                     # "May"

t1[0] = "JAN"
print(t1[0])                      # "JAN"

t2[0] = "sneezy"                  # raises TypeError
```

Tuples are useful for storing data that should not be changed during program execution ("const" behavior). Generally, treat a tuple as a coherent unit of data, like a date or an address.

List Slicing

Very handy: you can get arbitrary sub-lists from a list using slicing. The syntax is `list[start:stop:step]`. For example:

```
x = range(0,8)

print(x)              # [0, 1, 2, 3, 4, 5, 6, 7]

print(x[1:])          # [1, 2, 3, 4, 5, 6, 7]
print(x[:-1])         # [0, 1, 2, 3, 4, 5, 6]

print(x[1:6])         # [1, 2, 3, 4, 5]
print(x[1:6:2])       # [1, 3, 5]

print(x[::-1])        # [7, 6, 5, 4, 3, 2, 1, 0]
```

Use slicing to easily reverse a list or step through the indices.

The For Loop

Iterating Through a List

The `for` loop is used to iterate through a list and do something to each of its values:

```
xr = range(0,8)      # [0, 1, 2, 3, 4, 5, 6, 7]

y = []
for x in xr:
    print(x)
    y.append(x**2)   # stuff x^2 into y

print(y)             # [0, 1, 4, 9, 16, 25, 36, 49]
```

Lists are dynamic objects which can grow as a program executes. In this example we use `for` to loop through a list, square each of its values, and stuff the result into a second list using the `append` function.

Table of Contents

NumPy Arrays

Extensions to Lists

Using the `for` loop every time you want to do something to a list gets pretty annoying. Luckily the `numpy` module defines a type called `array` which supports *vectorized* arithmetic:

```
import numpy as np

x = np.arange(0,8)  # arange is like range()

y = x**2            # acts like a for loop over x
print(y)            # [0, 1, 4, 9, 16, 25, 36, 49]
```

In this example the single line `y = x**2` replaces 3 lines of code from the previous example, make the script more readable and less error-prone.

NumPy Arrays

Array Creation Routines

NumPy comes with a large number of creation routines [3], such as:

- An array of arbitrary shape initialized with zero:

```
>>> np.zeros(10, dtype=float)
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

- An array of evenly spaced numbers on a log scale:

```
>>> np.logspace(1, 6, 6, base=2)
array([  2.,   4.,   8.,  16.,  32.,  64.])
```

- An $N \times N$ identity matrix:

```
>>> np.identity(2)
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Slicing NumPy Arrays with Boolean "Masks"

A "mask" array can select values which satisfy a logical condition:

```
import numpy as np

x = np.arange(0,8)   # [0, 1, 2, 3, 4, 5, 6, 7]
y = 3*x              # [0, 3, 6, 9, 12, 15, 18, 21]

c = x < 3            # An array of Boolean values:
print(c)             # [True, True, True, False,
                     # [False, False, False, False]

print(x[c])          # selects [0, 1, 2]

c = (x<3) | (x>5)    # Combine cuts w/ bitwise OR (|)
print(y[c])          # select [0, 3, 6, 18, 21]
```

This is the type of selection used *all the time* in data analysis.

Multidimensional Arrays

Arrays of arrays (a.k.a. multidimensional arrays) can be used to represent matrices:

```
import numpy as np

# Make A a 3x3 matrix
A = np.arange(1,9).reshape((3,3))
print(A)                    # [[1, 2, 3],
                            #  [4, 5, 6],
                            #  [7, 8, 9]]

# Transpose
                            # [[1, 4, 7],
B = A.T                     #  [2, 5, 8],
                            #  [3, 6, 9]]
```

All of the usual matrix operations are supported.

Linear Algebra

Linear algebra is supported with the `numpy.linalg` module:

```
import numpy as np                      # A = [[4, 2],
                                        #      [2, 5]]
A = np.array([[4, 2], [2, 5]])
v, U = np.linalg.eig(A)                 # v = eigenvalues
                                        # U = eigenvectors
D = np.dot(np.dot(U.T, A), U)
```

The function `linalg.eig` calculates the eigenvalues and eigenvectors of the real symmetrix 2×2 matrix $\mathbf{A}$. We then diagonalize $\mathbf{A}$ by calculating the product

$$\mathbf{D} = \mathbf{U}^{\mathsf{T}} \cdot \mathbf{A} \cdot \mathbf{U},$$

where $\mathbf{U}$ is the matrix of eigenvectors.

File Input/Output

- File input and output is explained in the Python documentation.

```
f = open("filename.txt", "r")  # read-only access
for line in f:
    tokens = line.split()      # tokenize (string) data
    ...                        # cast strings to float...
```

- Better option: use the NumPy function `loadtxt` which reads data from a text file and loads it into a multidimensional table:

```
import numpy as np

data = np.loadtxt("filename.txt")   # load table
x = data[:,0]                       # x = column 0
y = data[:,1]                       # y = column 1
...
```

Plotting with Matplotlib

Plotting is trivial with the matplotlib package. My recommendation: go to the matplotlib gallery to learn how to make different kinds of plots.

```
%matplotlib inline      # needed in IPython notebook only

import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt("file.txt")   # load table
x = data[:,0]                   # x = column 0
y = data[:,1]                   # y = column 1

plt.plot(x, y, "k.")            # make an xy plot
plt.xlabel("x [arb. units]")
plt.ylabel("y [arb. units]")    # don't forget labels!
plt.title("Some XY data")
plt.show()                      # not needed in notebook
```

Table of Contents

Help Manual and Inspection

When running interactive sessions you can use the built-in `help` function to print module and function documentation.

```
In [1]: from fractions import gcd

In [2]: help(gcd)    # Opens a help dialog.
                     # Type Q to quit.

Help on function gcd in module fractions:

gcd(a, b)
    Calculate the Greatest Common Divisor of a and
    b.

    Unless b==0, the result will have the same sign
    as b (so that when b is divided by it, the
    result comes out positive).
```

Help Manual and Inspection

The `inspect` module can be used to view the source code of functions imported into your python session.

```
In [1]: from fractions import gcd
In [2]: import inspect
In [3]: print(inspect.getsource(gcd))
def gcd(a, b):
    """Calculate the Greatest Common Divisor of a
    and b.

    Unless b==0, the result will have the same sign
    as b (so that when b is divided by it, the
    result comes out positive).
    """
    while b:
        a, b = b, a%b
    return a
```

The Python Profiler

When calculations get large, efficiency starts to become important. Python has several diagnostic tools available to check for bottlenecks in program execution [5]. The easiest to use is the internal profiler (`prun`). Here is the result for the recursive Fibonacci function:

```
%prun fib(20)

         21893 function calls (3 primitive calls) in 0.005 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.005    0.005 fibslow.py:8(<module>)
  21891/1    0.005    0.000    0.005    0.005 fibslow.py:8(fib)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
      '_lsprof.Profiler' objects}
```

The function call `fib(20)` results in 21981 calls to `fib`. We weren't kidding when we said it was inefficient.

References I

[1] Andrew Binstock. *The Rise and Fall of Languages in 2013*. Jan. 2014. URL: http://www.drdobbs.com/jvm/the-rise-and-fall-of-languages-in-2013.

[2] Open HUB. *Compare Languages: Monthly Commits*. Dec. 2014. URL: https://www.openhub.net/languages/compare.

[3] *NumPy Array Creation Routines*. URL: http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html.

[4] Fernando Pérez and Brian E. Granger. "IPython: a System for Interactive Scientific Computing". In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. URL: http://ipython.org.

[5] *Python Speed*. URL: https://wiki.python.org/moin/PythonSpeed.

References II

[6] *Python Tutorial: 7. Input and Output*. URL: https://docs.python.org/2/tutorial/inputoutput.html.