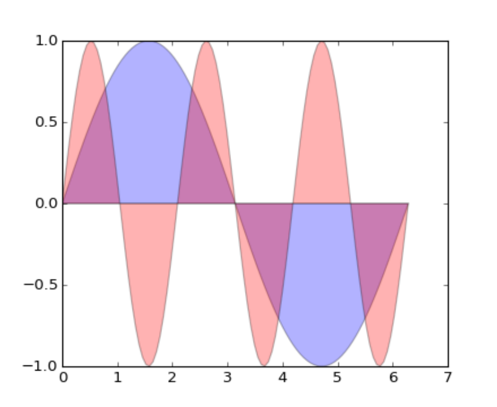
LECTURE 22

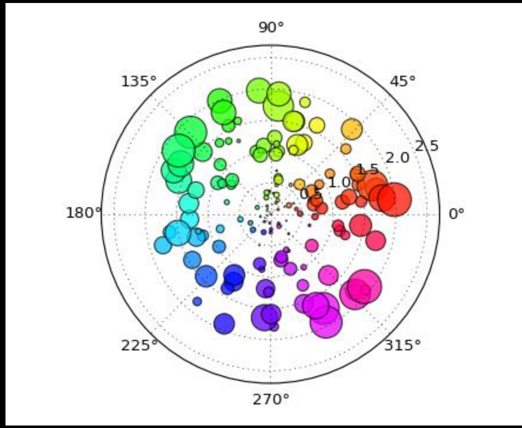
Numerical and Scientific Computing Part 2

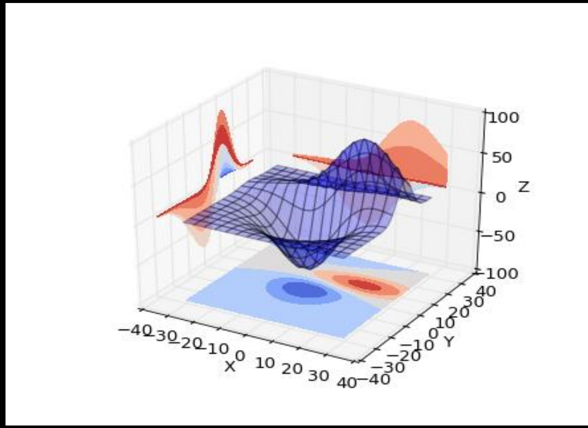
MATPLOTLIB

We're going to continue our discussion of scientific computing with matplotlib.

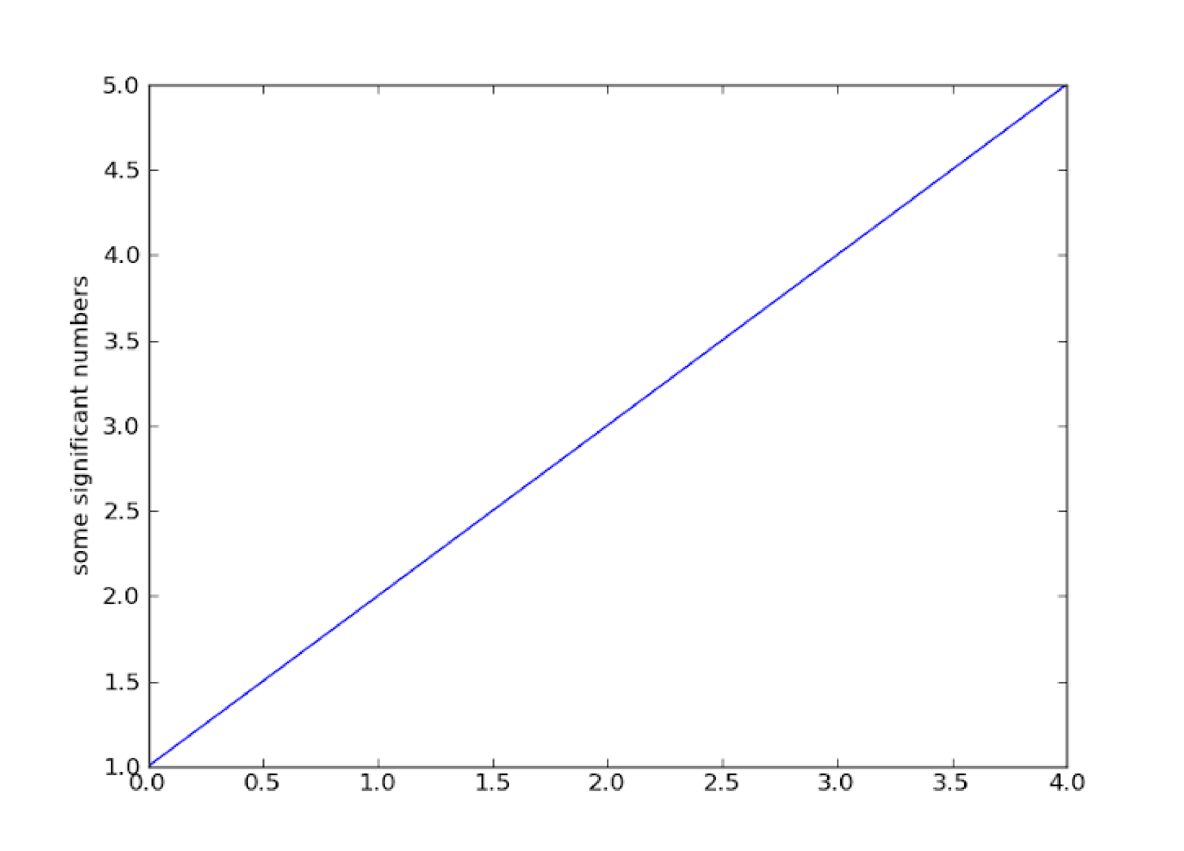
Matplotlib is an incredibly powerful (and beautiful!) 2-D plotting library. It's easy to use and provides a huge number of examples for tackling unique problems.







PYPLOT



At the center of most matplotlib scripts is pyplot. The pyplot module is stateful and tracks changes to a *figure*. All pyplot functions revolve around creating or manipulating the state of a figure.

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4,5])
plt.ylabel('some significant numbers')
plt.show()
```

When a single sequence object is passed to the plot function, it will generate the x-values for you starting with 0.

PYPLOT

The plot function can actually take any number of arguments. Common usage of plot:

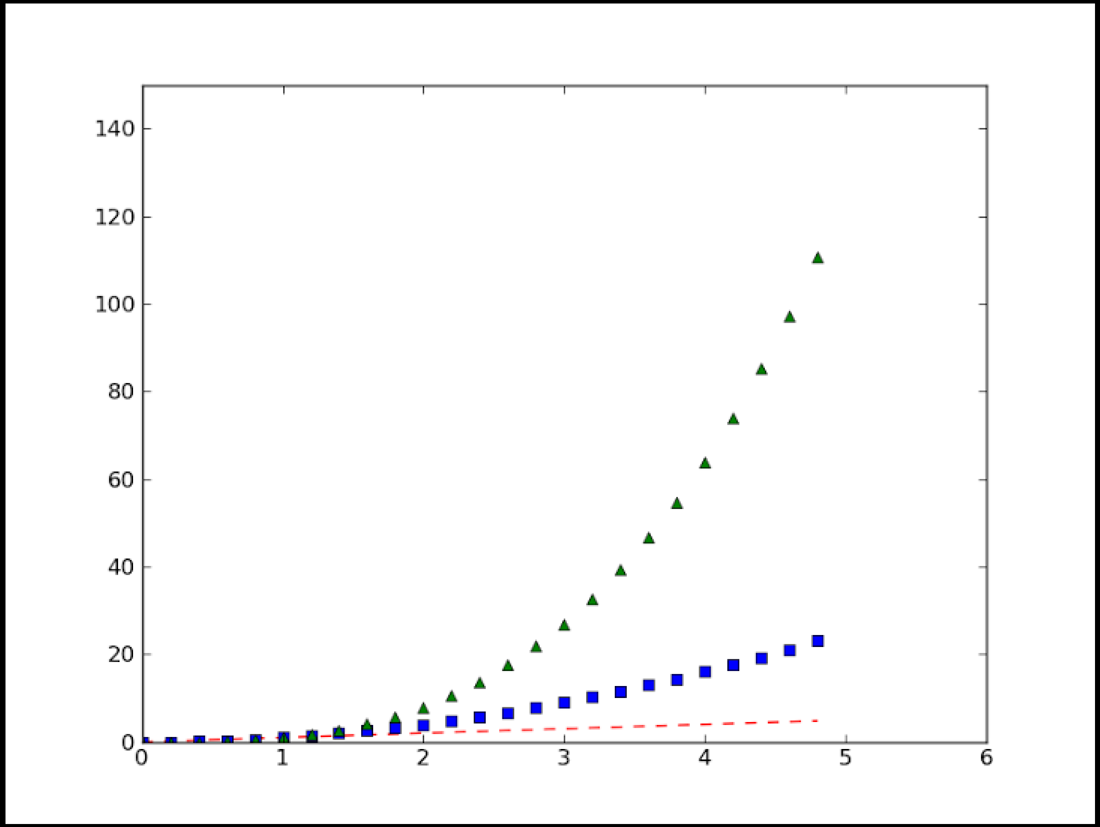
```
plt.plot(x_values, y_values, format_string [, x, y, format, ])
```

The format string argument associated with a pair of sequence objects indicates the color and line type of the plot (e.g. ‘bs’ indicates blue squares and ‘ro’ indicates red circles).

Generally speaking, the `x_values` and `y_values` will be numpy arrays and if not, they will be converted to numpy arrays internally.

Line properties can be set via keyword arguments to the plot function. Examples include `label`, `linewidth`, `animated`, `color`, etc…

PYPLOT




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

# evenly sampled time at .2 intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.axis([0, 6, 0, 150]) # x and y range of axis
plt.show()
```

BEHIND THE SCENES

It's important to note that a figure is a separate idea from how it is rendered. Pyplot convenience methods are used for creating figures and immediately displaying them in a pop up window. An alternative way to create this figure is shown below.

```
import numpy as np
import matplotlib.figure as figure

t = np.arange(0, 5, .2)

f = figure.Figure()
axes = f.add_subplot(111)
axes.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
axes.axis([0, 6, 0, 150])
```

PYPLOT

A script can generate multiple figures, but typically you'll only have one.

To create multiple plots within a figure, either use the `subplot()` function which manages the layout of the figure or use `add_axes()`.

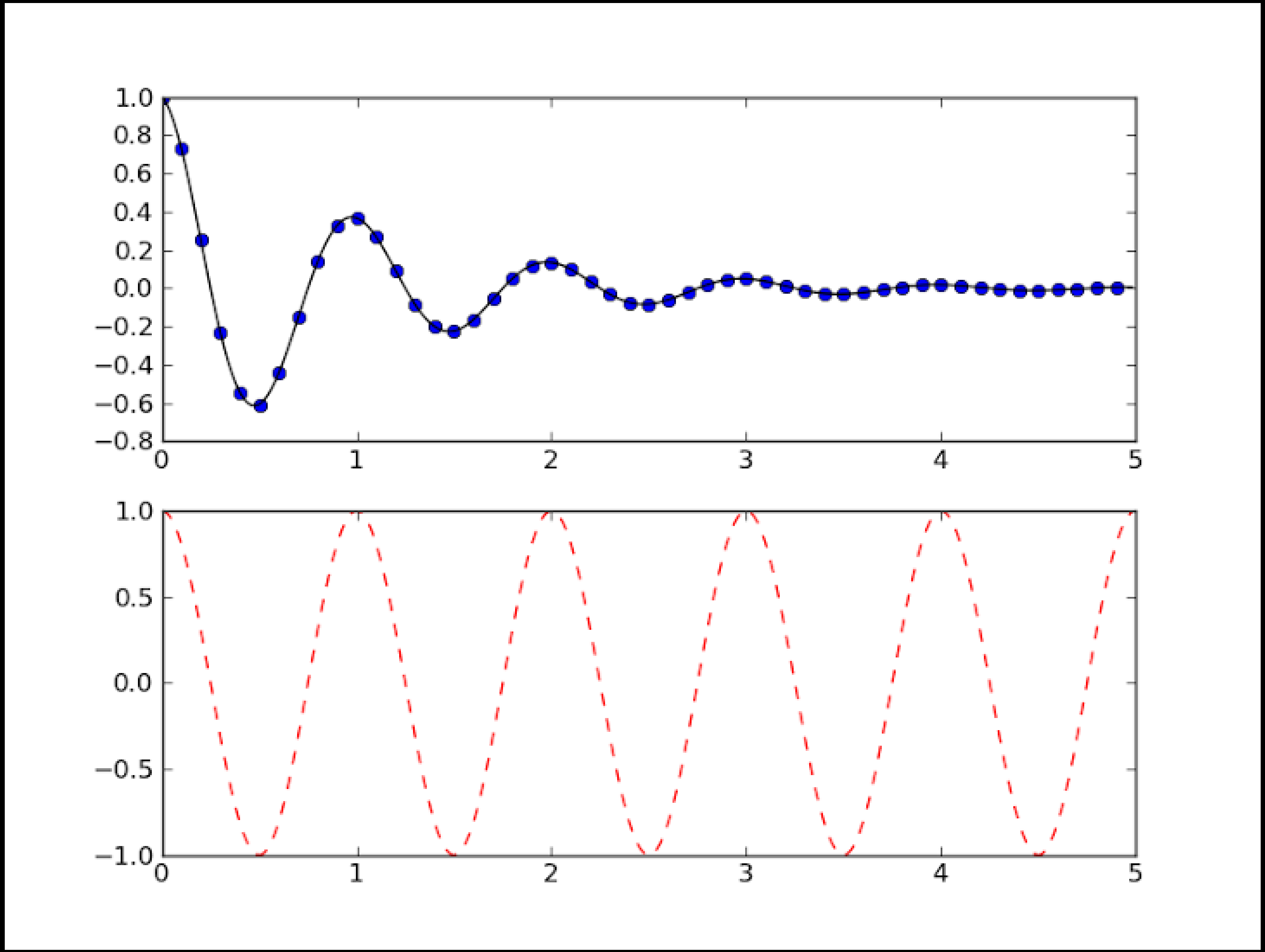
```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
plt.figure(1) # Called implicitly but can use
              # for multiple figures
plt.subplot(211) # 2 rows, 1 column, 1st plot
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212) # 2 rows, 1 column, 2nd plot
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```

PYPLOT



PYPLOT

- The `text()` command can be used to add text in an arbitrary location
- `xlabel()` adds text to x-axis.
- `ylabel()` adds text to y-axis.
- `title()` adds title to plot.
- `clear()` removes all plots from the axes.

All methods are available on pyplot and on the axes instance generally.

PYPLOT

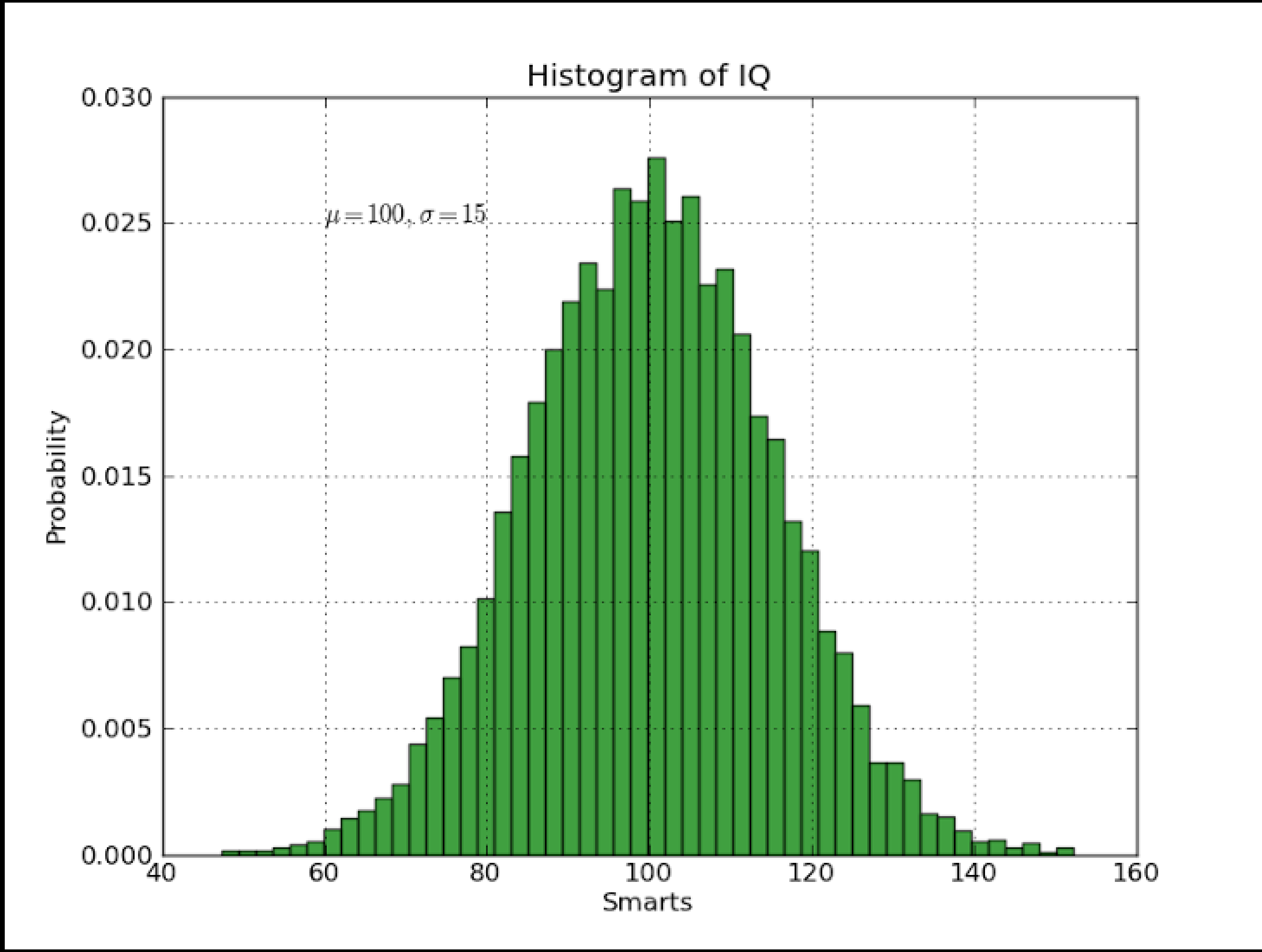
```
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$') #TeX equations
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

PYPLOT


Histogram of IQ
$\mu=100,\ \sigma=15$
Probability
Smarts
0.000
0.005
0.010
0.015
0.020
0.025
0.030
40
60
80
100
120
140
160

PYPLOT

There are tons of specialized functions – check out the API here. Also check out the examples list to get a feel for what matploblib is capable of (it's a lot!).

You can also embed plots into GUI applications.

For PyQt4, use `matplotlib.backends.backend_qt4agg`.

Let's do a little demonstration.

PLOT GUI

The two most important classes from matplotlib.backends.backend_qt4agg:

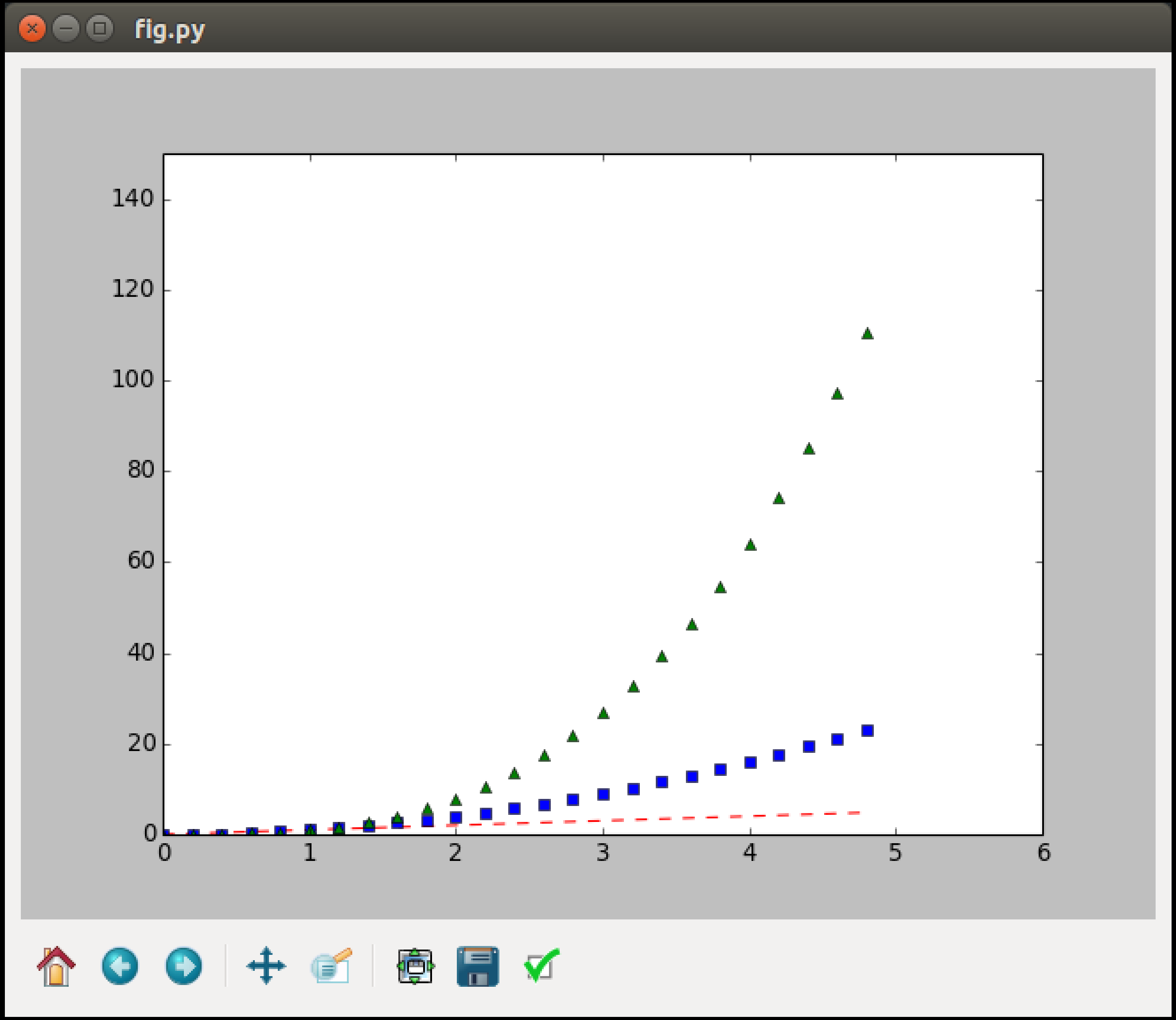
- `FigureCanvasQTAgg(`*`fig`*`)`: returns the canvas the figure *fig* renders into.
- `NavigationToolbar2QT(`*`canvas, prnt`*`)`: creates a navigation toolbar for *canvas* which has the parent *prnt*.

Furthermore, a canvas object has the following method defined:

- `canvas.draw()`: redraws the updated figure on the canvas.

PLOT GUI

Let's look at a little example using some plotting code from earlier. Check out plot_test.py.



IPYTHON

IPython stands for Interactive Python.

IPython was developed out of the desire to create a better interactive environment than the built-in Python interpreter allows.

- Interactive shell: `ipython`
- Interactive shell with PyQt GUI: `ipython qtconsole`
- Notebook server: `ipython notebook`

To install IPython, simply issue `sudo pip install ipython`.

IPYTHON

To start with, the IPython shell can be used just like the Python interpreter.

```
IPython 3.0.0 -- 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]: print "Hello, World!"
Hello, World!

In [2]: import sum_squares

In [3]: sum_squares.sum_of_squares(100)
Out[3]: 338350

In [4]: 2 + 7*3
Out[4]: 23
```

IPYTHON

To start with, the IPython shell can be used just like the Python interpreter.

We can execute python statements one at a time as well as import modules and call functions just like we can with the Python interpreter.

Note the interface is a series of inputs to and outputs from the shell.

IPython's goal, however, is to provide a superior interpreted environment for Python so there's a lot more we can do.

```
In [1]: print "Hello, World!"
Hello, World!

In [2]: import sum_squares

In [3]: sum_squares.sum_of_squares(100)
Out[3]: 338350

In [4]: 2 + 7*3
Out[4]: 23
```

IPYTHON REFERENCE

Whenever you start up IPython, you'll get the following message:

```
?         -> 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.
```

The first will basically show you IPython's quick documentation and the second gives you a rundown of a lot of the functionality. For either, type 'q' to go back to the shell.

IPYTHON HELP

Whenever you start up IPython, you'll get the following message:

```
?          -> 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.
```

The help command can be used to either start up a separate help interpreter (just type `help()`) or get help for a specific object.

```
In [13]: x = 3

In [14]: help(x)
```

IPYTHON OBJECT INSPECTION

The object? command can be used to get some more details about a particular object. IPython refers to this as *dynamic object introspection*. One can access docstrings, function prototypes, source code, source files and other details of any object accessible to the interpreter.

```
In [16]: l = [1,2,3]
In [17]: l?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

IPYTHON TAB COMPLETION

Tab completion: typing tab after a python object will allow you to view its attributes and methods. Tab completion can also be used to complete file and directory names.

```
In [4]: sum_
sum_squares      sum_squares.py   sum_squares.pyc  sum_squares.py~

In [4]: sum_squares.s
sum_squares.square_of_sum   sum_squares.sum_of_squares

In [4]: sum_squares.
sum_squares.print_function  sum_squares.py~
sum_squares.py              sum_squares.square_of_sum
sum_squares.pyc             sum_squares.sum_of_squares
```

IPYTHON MAGIC COMMANDS

IPython supports "magic commands" of the form `%command`.

These commands can be used to manipulate the IPython environment, access directories, and make common system shell commands.

- *Line-oriented* magic commands accept as arguments whatever follows the command, no parentheses or commas necessary.
  - Examples: `%run`, `%timeit`, and `%colors`
- *Cell-oriented* magic commands (prefixed with %%) take not only the rest of the line, but also subsequent lines as arguments.
  - Examples: `%%writefile`, `%%timeit`, `%%latex`

IPYTHON MAGIC COMMANDS

The %timeit command comes in both line-oriented and cell-oriented form.

```
In [1]: %timeit range(1000)
100000 loops, best of 3: 13.2 us per loop

In [2]: %%timeit x = range(10000)
   ...: max(x)
   ...:
1000 loops, best of 3: 287 us per loop
```

Optionally:

- `-n <N>` Execute the given statement <N> times in a loop.
- `-r <R>` Repeat the loop iteration <R> times and take the best result.
- `-o` Return a TimeitResult that can be stored in a variable.

IPYTHON MAGIC COMMANDS

The `%%script` command allows you to specify a program and any number of lines of script to be run using the program specified.

```
In [1]: %%script bash
   ...: for i in 1 2 3; do
   ...:   echo $i
   ...: done
1
2
3
```

Optionally:

`--bg`	Whether to run the script in the background.
`--err ERR`	The variable in which to store stderr from the script.
`--out OUT`	The variable in which to store stdout from the script.

IPYTHON RUN COMMAND

The magic command %run is especially useful: it allows you to execute a program within IPython.

The IPython statement

```
In [1]: %run file.py args
```

is similar to the command

```
$ python file.py args
```

but with the advantage of giving you IPython's tracebacks, and of loading all variables into your interactive namespace for further use

PYTHON RUN COMMAND

```python
print "Hello from test.py!"

def f(x):
    print "Argument is ", x

if __name__ == "__main__":
    print "Calling f(3)."
    f(3)
```

Consider the python program test.py to the right.

```
In [1]: %run test.py
Hello from test.py!
Calling f(3).
Argument is  3

In [2]: %run -n test.py
Hello from test.py!
```

By default, programs are run with __name__ having the value "__main__". The –n flag indicates that __name__ should have the module's name instead.

PYTHON RUN COMMAND

Consider the python program test.py to the right.

```
In [1]: y = 4

In [2]: %run -i test.py
Hello from test.py!
Calling f(3).
Argument is  3
The variable y is  4
```

```python
print "Hello from test.py!"

def f(x):
    print "Argument is ", x
    print "The variable y is ", y


if __name__ == "__main__":
    print "Calling f(3)."
    f(3)
```

The –i flag indicates that we should use IPython’s namespace instead of a brand new empty one.

PYTHON RUN COMMAND

Consider the python program test.py to the right.

```
In [1]: %run -i -t -N100 test.py

IPython CPU timings (estimated):
Total runs performed: 100
  Times  :      Total       Per run
  User   :       8.81 s,        0.09 s.
  System :       0.04 s,        0.00 s.
Wall time:       8.84 s.
```

```python
def f(x):
    total = x
    for i in range(1000000):
        total += i
if __name__ == "__main__":
    f(3)
```

The –t flag indicates that we would like to time the execution of the module N times.

PYTHON RUN COMMAND

Consider the python program test.py to the right.

```
In [1]: %run test.ipy
Type:          int
String form: 4953
Docstring:
int(x=0) -> int or long
int(x, base=10) -> int or long
```

Running a .ipy or .nb file indicates that we'd like the contents to be interpreted as ipython commands.

test.py

```
total = 0

def f(x):
    global total
    total = x
    for i in range(100):
        total += i

if __name__ == "__main__":
    f(3)
```

test.ipy

```
%run -i test.py
total?
```

PYTHON EDIT COMMAND

The %edit command allows us to bring up an editor and edit some code.

The resulting code is immediately executed in the IPython shell.

```
In [15]: %edit
IPython will make a temporary file named:
/tmp/ipython_edit_qHYbnt/ipython_edit_mB8nlR.py
Editing... done. Executing edited code...
Out[15]: 'def say_hello():\n    print "Hello, World!"\n'

In [17]: say_hello()
Hello, World!

In [18]: %edit say_hello
Editing... done. Executing edited code...

In [19]: say_hello()
Hello again, world!
```

IPYTHON HISTORY

One of the most convenient features offered by IPython is the ability to save and manipulate the interpreter's history.

Up- and down- arrow keys allow you to cycle through previously-issued commands, even across IPython sessions.

Previously issued commands are available in the `In` variable, while previous output is available in the `Out` variable.

```
In [17]: say_hello()
Hello, World!

In [18]: edit say_hello
Editing... done. Executing edited code...

In [19]: say_hello()
Hello again, world!

In [20]: In[17]
Out[20]: u'say_hello()'

In [21]: exec(In[17])
Hello again, world!
```

IPYTHON HISTORY

The last three objects in output history are also kept in variables named _, __ and ___. A number of magic commands also accept these variables (e.g. `%edit _`).

The `%history` magic command will also list some previous commands from your history.

```
In [25]: 2+3
Out[25]: 5

In [26]: _ + 4
Out[26]: 9

In [27]: _ + __
Out[27]: 14
```

IPYTHON SHELL

You can run any system shell command within IPython by prefixing it with a !.

```
In [32]: !ls ticket_app/app/
forms.py __init__.py  static  templates  ticket_scraper.py  views.py

In [33]: files = !ls ticket_app/app/

In [34]: print files
['forms.py', '__init__.py', 'static', 'templates',
'ticket_scraper.py', 'views.py']
```

IPYTHON NOTEBOOK

So, now we've seen IPython's awesome command shell. Whether you choose terminal or Qt-based, it's definitely a great step-up from the built-in Python interpreter.

IPython Notebook, however, provides an entirely new type of environment for developing, executing, and displaying Python programs.

IPython Notebook is a web-based interactive computational environment for creating IPython notebooks. An IPython notebook is a JSON document containing an ordered list of input/output cells which can contain code, text, mathematics, plots and rich media.

Check out the Notebook gallery.

IPYTHON NOTEBOOK

IPython Notebook has two components:

- Web application: you can start an IPython notebook server for writing documents that contain IPython tools.
- Notebook documents: a computational record of a session within the IPython notebook server.

IPYTHON NOTEBOOK

Start a notebook server with the following command:

```
$ ipython notebook
```

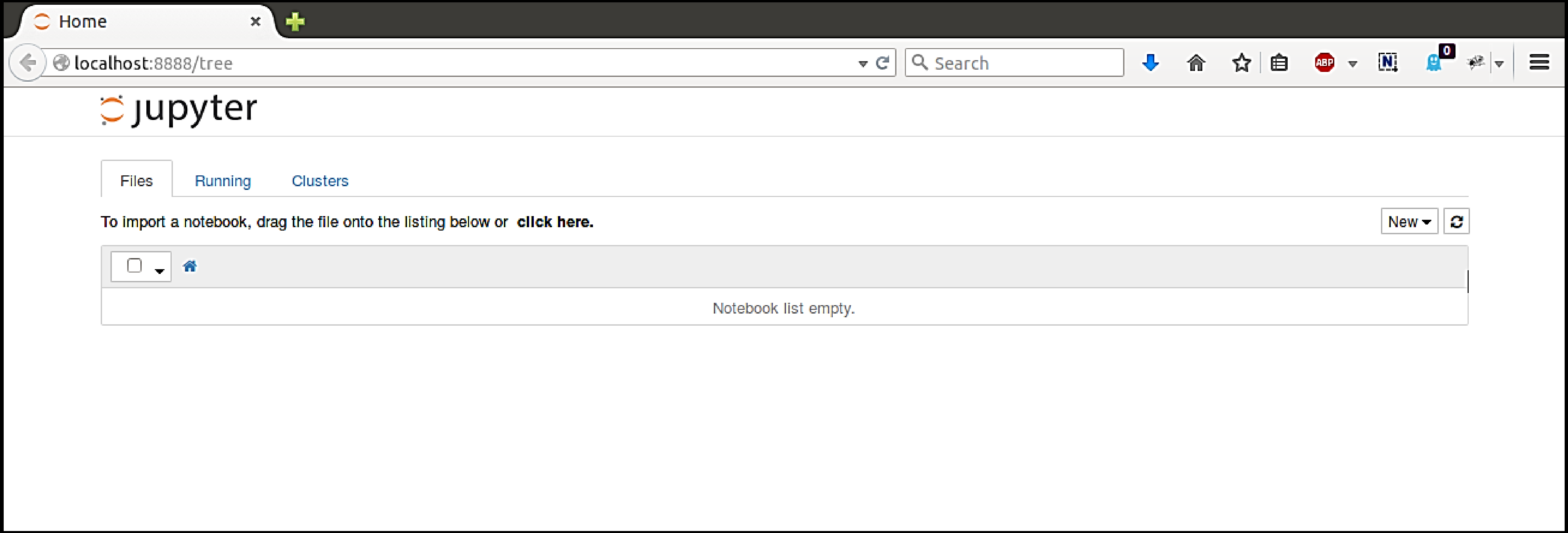
This will print some information about the notebook server in your console, and open a web browser to the URL of the web application (by default, http://127.0.0.1:8888).

The page you open to will display a listing of the notebooks in your current directory.

Additionally, you can specify that a particular notebook should be opened when the application is launched:

```
$ ipython notebook some_notebook.ipynb
```

IPYTHON NOTEBOOK



IPYTHON NOTEBOOK

Initializing an IPython server means that you are initializing an *IPython kernel*.

Traditionally, interpreters use a REPL format to communicate with the user. That is, the user enters a statement which is read, evaluated, and the output of which is printed to the screen. This loop continues until the user quits the interpreter.

Almost all aspects of IPython, however, use a unique two process model where every interactive session is really a client connection to a separate evaluation process, of which the IPython kernel is in charge.

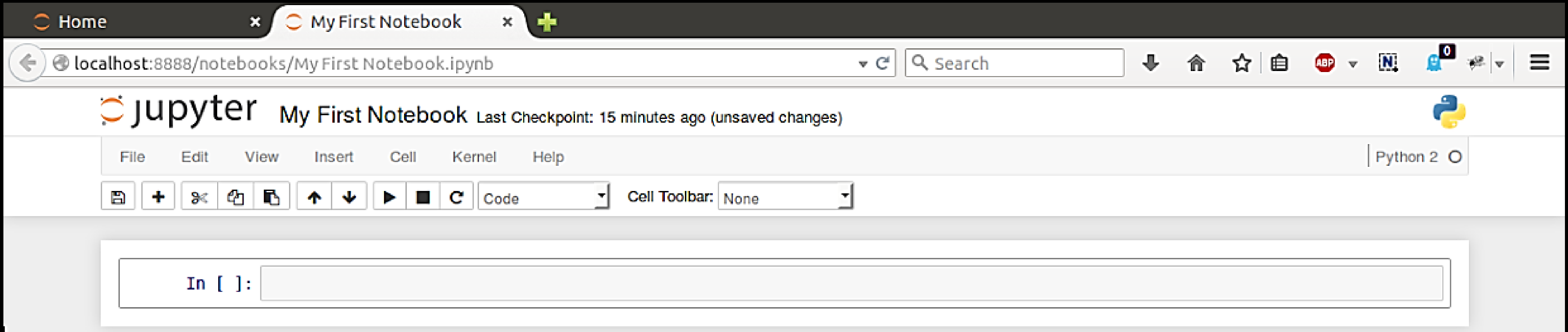
IPYTHON NOTEBOOK

So, what's the advantage of this two-process model?

- Several clients can connect to the same kernel.
- Clients can connect to kernels on a remote machine.
- To connect to the most-recently created kernel:
  ```
  $ ipython notebook --existing
  ```
- To connect to a specific kernel:
  ```
  $ ipython notebook --existing kernel-19732.json
  ```
- Use the magic command `%connect_info` to get the connection file for a kernel.

IPYTHON NOTEBOOKS

Let's create an IPython notebook together.



The notebook consists of a sequence of cells. A cell is a multi-line text input field, and its contents can be executed by using Shift-Enter.

IPYTHON NOTEBOOK

There are four types of cells, each having different execution behavior:

Code cells – contain Python code by default, but other languagues as well. Output is not limited to text, but can also be figures and other rich media.

Markdown cells – contain Markdown code and render rich text output.

Raw cells – contain output directly. Raw cells are not evaluated by kernel.

Heading cells – contain one of 6 different heading levels to provide structure to document.

The default is code cell, but you can change this in the toolbar.

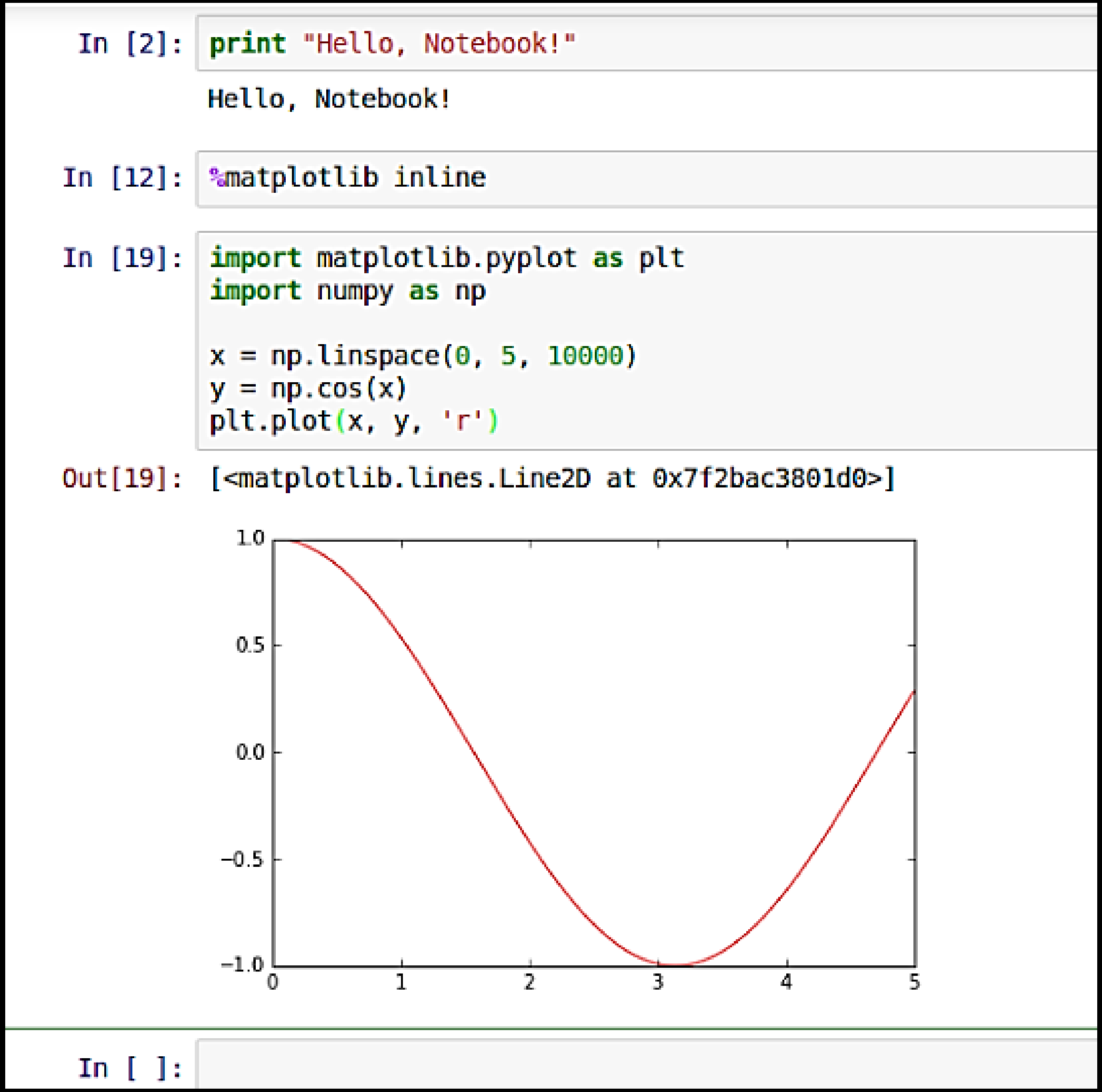
IPYTHON NOTEBOOK

Here are some basic code cells examples.

We can clearly write and execute typical Python code in our code cells.

But we can also write and execute code with rich output, like matplotlib plots.

Behind the scenes, the code of the cell is submitted to the kernel which evaluates the code and returns a response to the client application.



PYTHON NOTEBOOK

To display more exotic rich media, you'll need IPython's display module.

```
from IPython.display import display
```

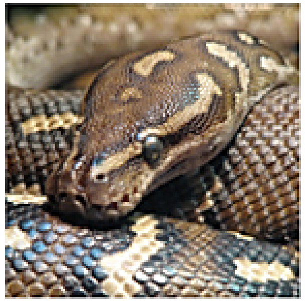
- `Image([`*`filename, url, …`*`])` creates a displayable image object.
- `SVG()` for displayable scalable vector graphics.
- `FileLink()` to create links.
- `HTML()`
- etc.

PYTHON NOTEBOOK

```
In [20]: from IPython.display import Image
         i = Image(filename = 'Python-4.jpg')

In [21]: i

Out[21]:
```

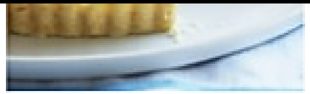


```
In [23]: i = Image(url = "http://recipedose.com/wp-content/uploads/2010/03/banoffee_pie-150x150.jpg")
         i

Out[23]:
```

```
In [ ]:
```

PYTHON NOTEBOOK



In [25]:
```
from IPython.display import Math
Math(r'i\hbar\frac{\partial }{\partial t}\Psi (\mathbf{r},t) = \left [ \frac{-\hbar^{2}}{2\mu }\bigtriangledown^{2} + V(\ma
```

Out[25]:

$$i\hbar\frac{\partial }{\partial t}\Psi (\mathbf{r},t) = \left [\frac{-\hbar^{2}}{2\mu }\bigtriangledown^{2} + V(\mathbf{r},t)\right] \Psi(\mathbf{r},t)$$

In [27]:
```
from IPython.display import HTML
HTML(r'<h1>This is a big header!</h1> <h3>And a little baby header!</h3>')
```

Out[27]:

This is a big header!

And a little baby header!

In [28]:
```
from IPython.display import FileLink
FileLink('My First Notebook.ipynb')
```

Out[28]: My First Notebook.ipynb

In [ ]: