Python for MATLAB Users

Promoting Open Source Computer Vision Research

Kitware, Inc. / Google Research

June 2012






Matt Leotta
Amitha Perera
Patrick Reynolds
Eran Swears
Yong Zhao
Varun Ganapathi

What this tutorial is about

- Promote open source computer vision research.
  - Reduce the community's dependence on MATLAB.
  - Show MATLAB users how to migrate to Python.
  - Present a Python IDE similar to that of MATLAB.
- Introduce Python for scientific computing.
- Explain how to access functionality in Python from third party vision libraries (OpenCV, ITK).
- Cover some advanced computing topics using Python:
  - parallel processing with threads
  - GPU processing with PyCUDA and PyOpenCL
  - calling C/C++ code from Python
  - calling legacy MATLAB code from Python
- Provided examples and hands-on exercises for all of the above.
- Raffle of an ASUS Transformer Pad and more.

Schedule (Approximate)

8:30	Overview, Virtual Machine Setup
9:00	Intro to Python
10:00–10:30	Coffee Break
10:30	Python Scripting
11:15	Scientific Python – Numpy, Matplotlib, SciPy
12:30–1:30	Lunch
1:30	Using OpenCV
2:15	Using Simple ITK
3:00–3:30	Coffee Break
3:30	Parallel Processing / GPU Processing
4:15	Interfacing Python with C/C++
4:45	Interfacing Python with MATLAB
5:00	Wrap-up, Raffle

Virtual Machine Distribution

Virtual Machine Distribution

- Get a USB Memory Stick
- Install VirtualBox from it
- Import the virtual machine file
- Boot the virtual machine
- Get familiar with the environment and directories

- VirtualBoxInstallers
  - VirtualBox-4.1.16-78094-Linux_amd64.run (64-bit Linux)
  - VirtualBox-4.1.16-78094-Linux_x86.run (32-bit Linux)
  - VirtualBox-4.1.16-78094-OSX.dmg (Mac)
  - VirtualBox-4.1.16-78094-Win.exe (Windows)
  - LinuxPackages
    - RPM (Fedora 14–17, Mandriva 2010–2011, RHEL/CentOS 4–6, SUSE 10–11, OpenSUSE 11.3–11.4)
    - DEB (Debian 5–7, Ubuntu 8.04, 10.04–12.04)
- VirtualMachine
  - matlab-to-python.ova

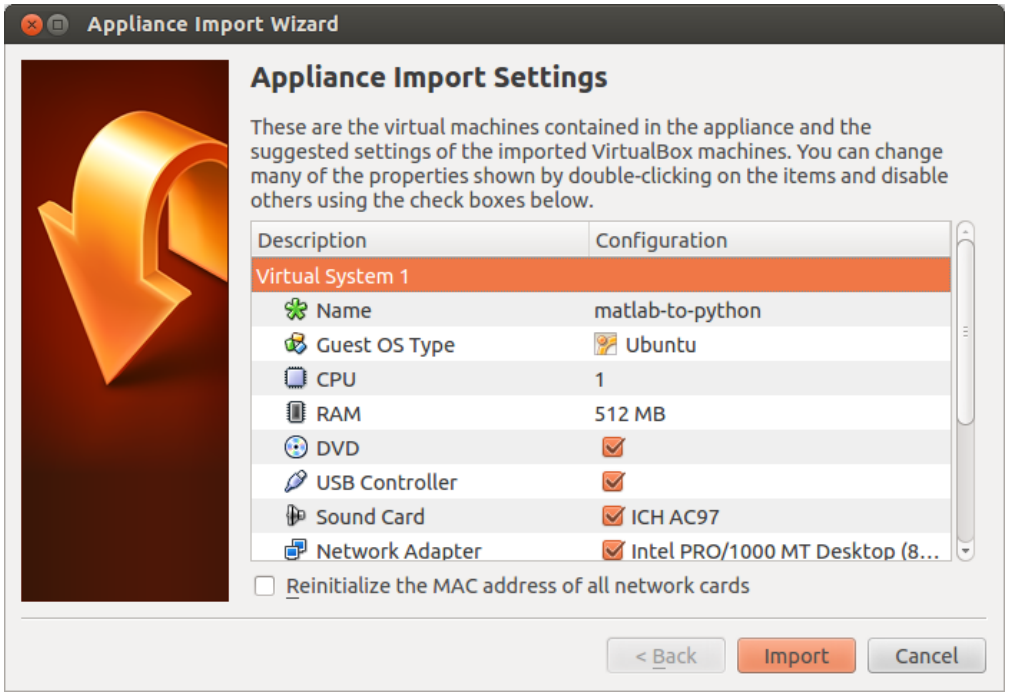
https://www.virtualbox.org/wiki/Downloads

Install VirtualBox

- Select the installer for your platform
- Run it

Importing the Virtual Machine

- Run VirtualBox
- In "File" Menu select "Import Appliance"
- Provide the filename in the USB stick "VirtualMachine/matlab-to-python.ova"
- You should see:



To Be Continued...

Why use Python?

Open Source Research

- Sharing code and data is often overlooked but is as important as sharing algorithms and results in published scientific research.
- Shared code promotes reproducible research and scientific validation.
- Computer vision research code is commonly written in MATLAB.
- Sharing research code written in MATLAB is a good first step, but . . .
- MATLAB code can only be executed by a privileged few who can afford MATLAB license fees.

Why not MATLAB?

- Language design
  - MATLAB is effective for quick code and/or small code bases, but does not scale as projects grow
  - Numerical computing is easy, but more general programing is not
- Top-level functions and code organization
  - Function-name = file-name in MATLAB: discourages writing modular programs
  - Hard to appropriately name functions and organize code without conflicts
- Toolboxes are purchased separately
  - Shared MATLAB code will only run if you've licensed the right toolboxes
- Cost – Very Expensive!

MATLAB Pricing

MATLAB Component	**Price**
Inidvidual Matlab License	$2,150
MATLAB Compiler	$5,000
Toolboxes	
Computer Vision System	$1,350
Image Processing	$1,000
Optimization	$1,000
Parallel Computing	$1,000
Statistics	$1,000
Neural Network	$1,000
Signal Processing	$1,000
Symbolic Math	$1,000
Total	$15,500 *per user*

- Pricing as of April 12th, 2012.

More Info: `http://www.mathworks.com/store/productIndexLink.do`

Open Source Alternatives to MATLAB

- Octave (GPL license)
  - A mostly MATLAB compatible language
  - Command line only (no GUI IDE)
- FreeMat (GPL license)
  - A somewhat MATLAB compatible language
  - Very basic GUI
- Scilab (CeCILL license)
  - Not compatible with MATLAB language, provides a converter
  - Most full featured IDE

- These options solve the cost and license restrictions, but have the same technical limitations MATLAB.
- None of these options are perfectly compatible with MATLAB code.
- Each has a relatively small user community.

Why Python?

- Cost / License
  - Python and related packages are free (have no licensing fee)
  - Python and related packages use permissive BSD-like licenses.
- Language design
  - Python has well-designed language features supporting small and large code bases
  - Python code is highly readable
- Top-level functions and code organization
  - Python is modular and supports name spaces to avoid naming conflicts.
- Integration with other languages/tools
- An enormous user base.
  - Python is used for just about everything (web, database, networking, GUI, gaming, graphics, science, . . .)
- Just search the web for testimonials

Interesting essay: `http://www.stat.washington.edu/~hoytak/blog/whypython.html`

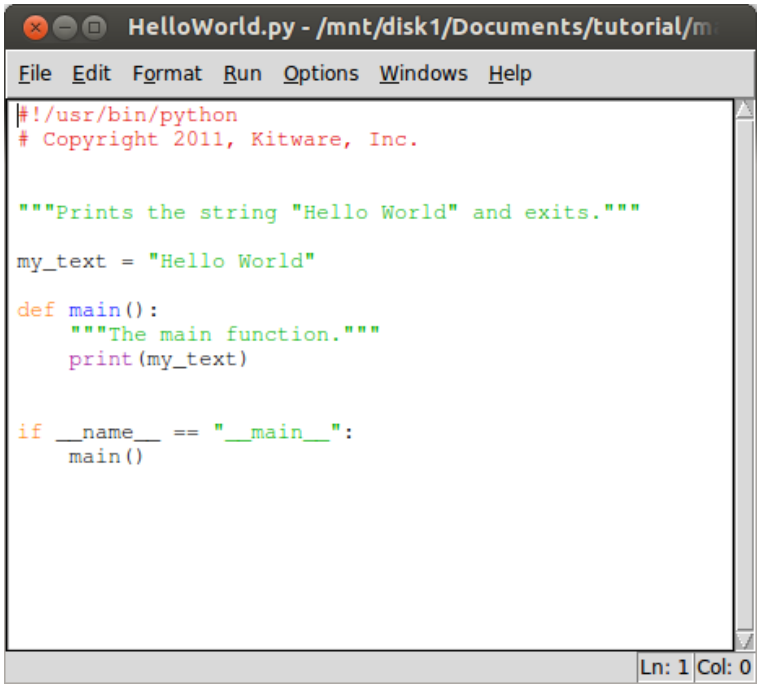
Ways to Use Python: Many IDE Choices

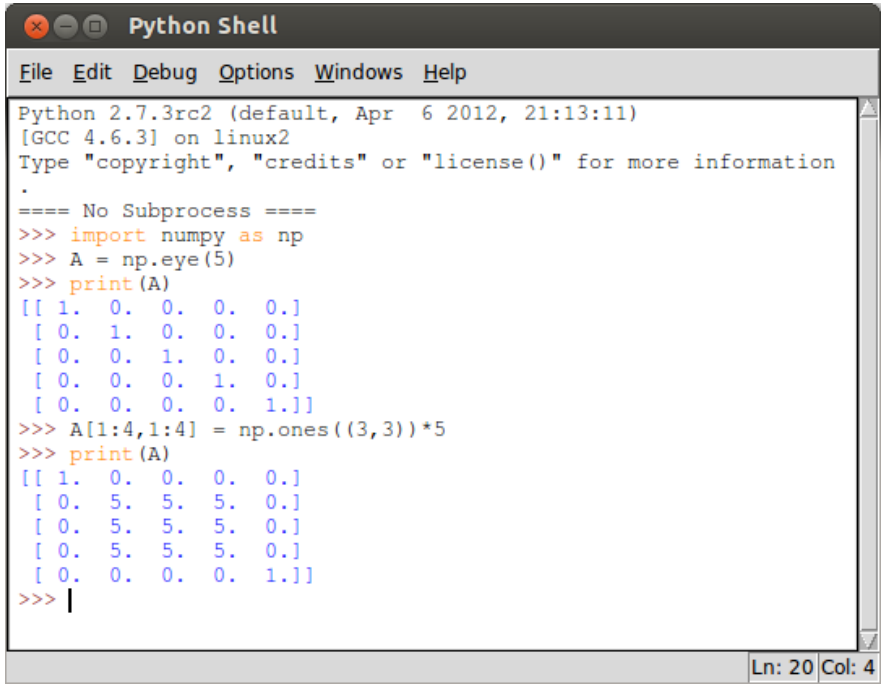
IDE	Developer	Latest stable release version	Latest stable release date	Platform	Toolkit	License
Anjuta	Naba Kumar	3.0.3.0	2011-06-10	Unix-like	GTK+	GPL
Eric Python IDE	Detlev Offenbach	4.4.18 and 5.1.6	2011-10-02	Independent	Qt	GPL
Geany	Team	0.20	2011-01-07	Independent	GTK2	GPL
IDLE	Guido van Rossum et al.	3.2	2011-05-13	Independent	Tkinter	Open source
Komodo IDE	ActiveState	6.0.0	2010-10-07	Cross-platform	Mozilla platform	Proprietary
MonoDevelop	Novell and the Mono community	2.4.2	2011-01-19	Cross-platform	Gtk#	LGPL
NetBeans (7.0 and up will no longer support Python)	Oracle	6.9	2010-06-15	Cross-platform	Swing	GPL
Ninja_IDE	Team	2.0-Beta3		Independent	Qt	GPL
PIDA	Team	0.6.2	2010-08-04	Cross-platform	PyGTK	GPL
PyCharm	JetBrains	2.0.2	2012-02-07	Cross-platform	Swing	Proprietary
PyDev	Aptana	2.4.0	2012-02-01	Eclipse (Cross-platform)	SWT	EPL
PyScripter		2.4.3	2011-09-20	Windows		MIT Licence
Python Tools for Visual Studio	Microsoft	1.1	2012-02-10	Windows		Apache License 2.0
Spyder	Pierre Raybaut et al.	2.1.9	2012-03-31	Independent	Qt	MIT Licence
Stani's Python Editor	Stani	0.8.4h	2008-02-14	Independent	wxPython	GPL
Wing	Wingware	4.1.3-1	2012-01-11	Linux/Windows/OS X	GTK2	Proprietary
wxGlade	Alberto Griggio	0.6.3	2008-02-02	Independent	wxPython	MIT License

- There are several choices for Python development environments.
- The above list is from Wikipedia (taken April 2012).
- Spyder is most like the MATLAB IDE.

http://en.wikipedia.org/wiki/List_of_integrated_devel...for_Python

Ways to Use Python: IDLE Python IDE

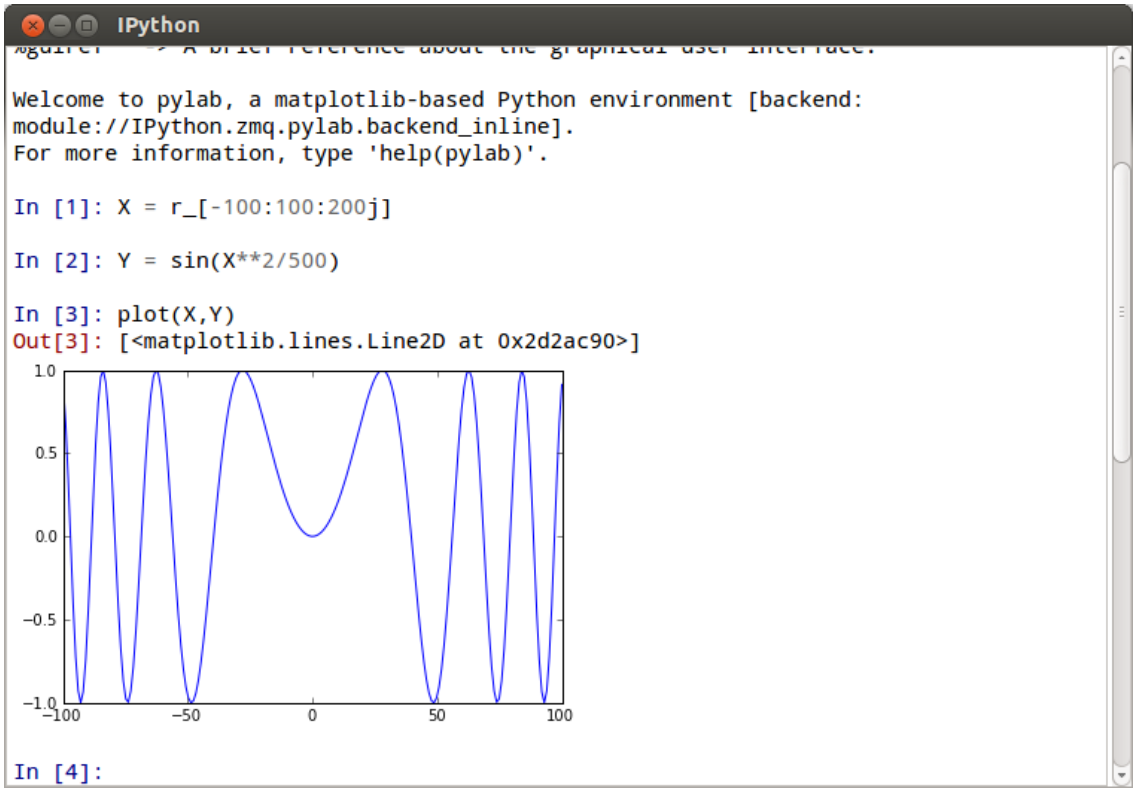




- The original IDE, bundled with most Python installations.
- IDLE is simple and not very intuitive to use.

More info: `http://docs.python.org/library/idle.html`

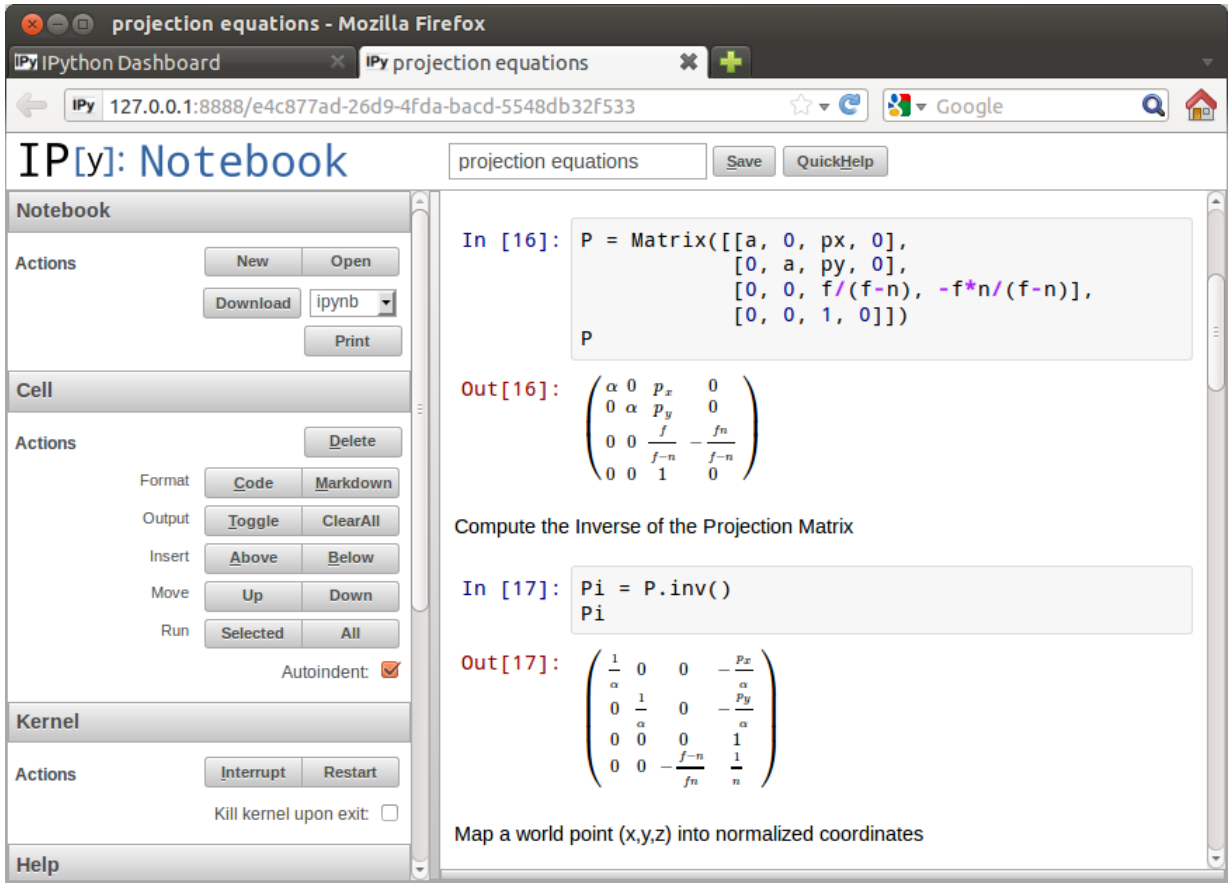
Ways to Use Python: IPython Qt Console



- IPython Qt Console allows for inline plotting.
- We will use IPython Qt Console to learn the basics of Python.

More info: `http://ipython.org/`

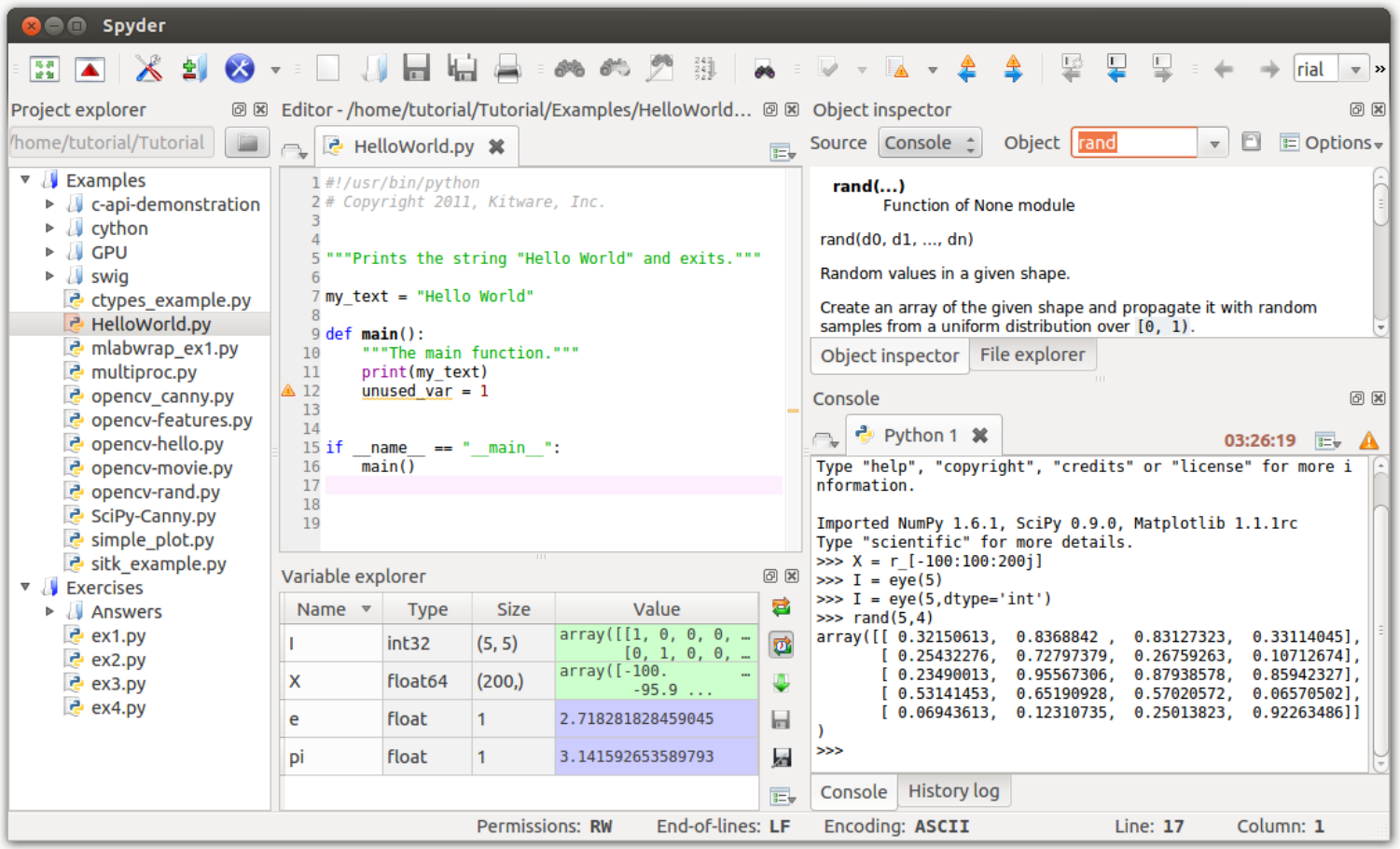
Ways to Use Python: SymPy with IPython Notebook



- IPython Notebook lets you keep a math notebook in a web browser.
- SimPy adds symbolic math for a Maple or Mathematica alternative.
- Run `ipython notebook --profile=sympy`

More info: `http://ipython.org/` and `http://sympy.org/`

Ways to Use Python: Spyder IDE



- Spyder provides an IDE similar to MATLAB.
- We will use Spyder later in this tutorial.

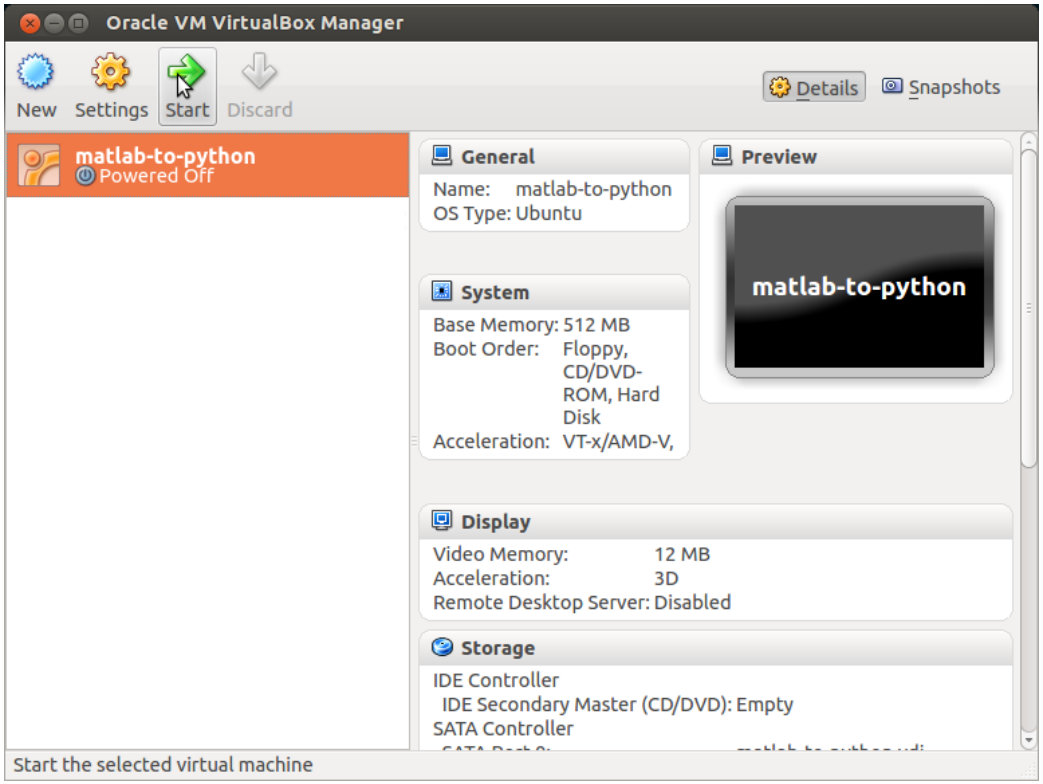
More info: `http://code.google.com/p/spyderlib/`

Using the Virtual Machine

Starting the Ubuntu 12.04 VM

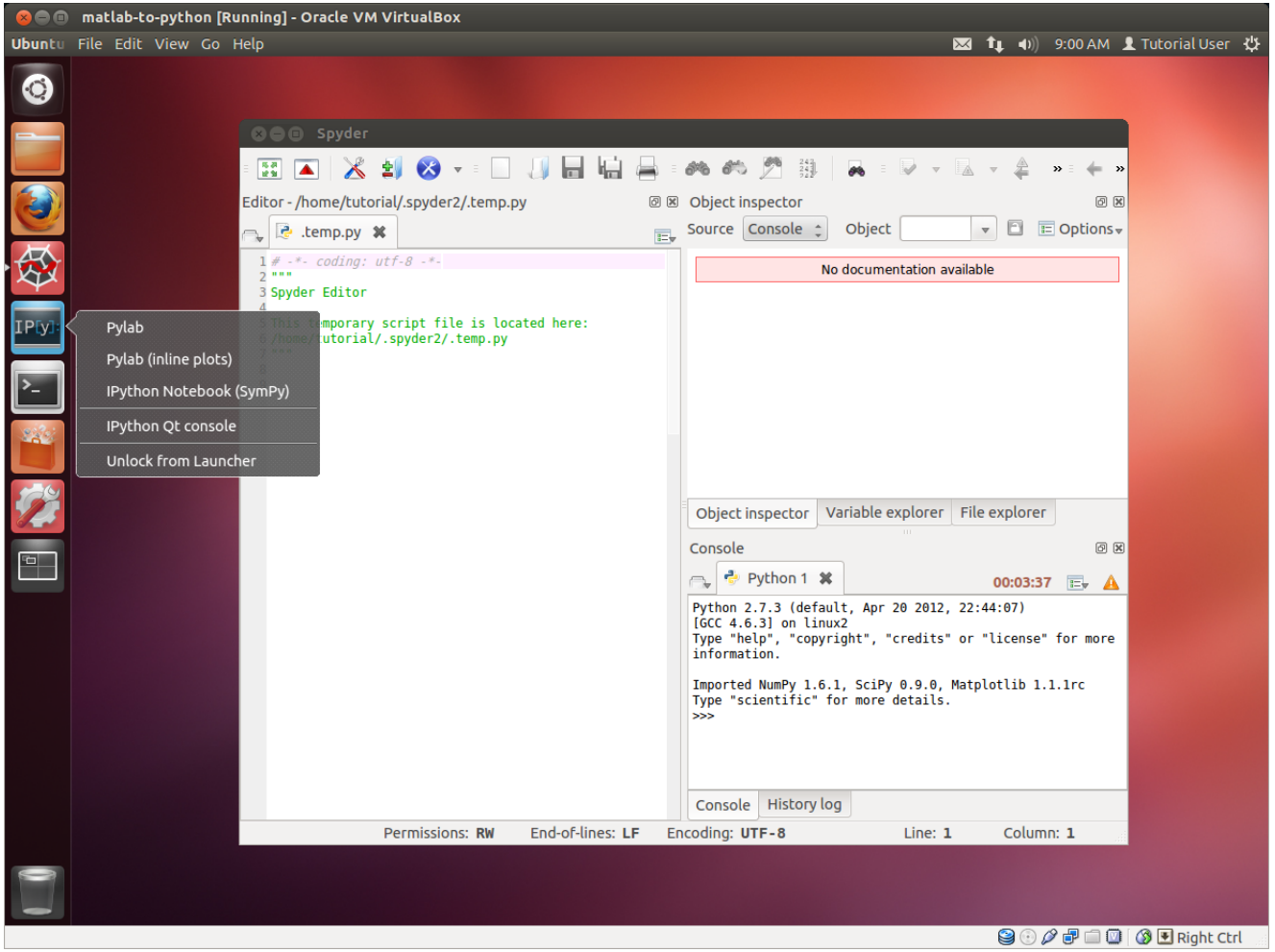
- Select the matlab-to-python VM.
- Press the "Start" button.
- Wait for the OS to boot.
- You will be automatically logged in as

  username tutorial
  password tutorial



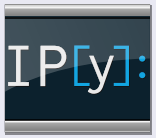
The Ubuntu 12.04 Desktop

- The bar on the left is the App launcher.
- Some Apps have additional options when you right click.
- Drop down menus are on the top of the screen (as in MacOS X).



Key Applications

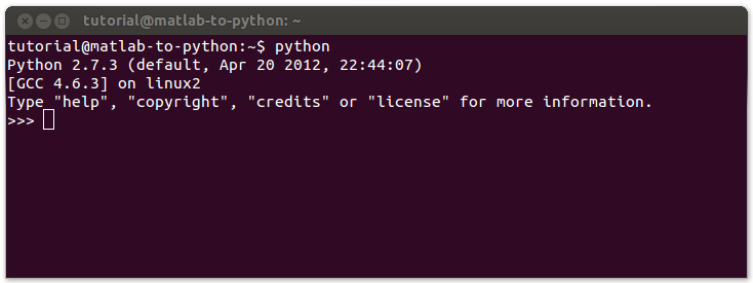
Key Applications

	Firefox	A web browser.
	Spyder	A MATLAB-like Python IDE.
	IPython	An enhanced interactive Python shell.
	Terminal	The Gnome terminal emulator.

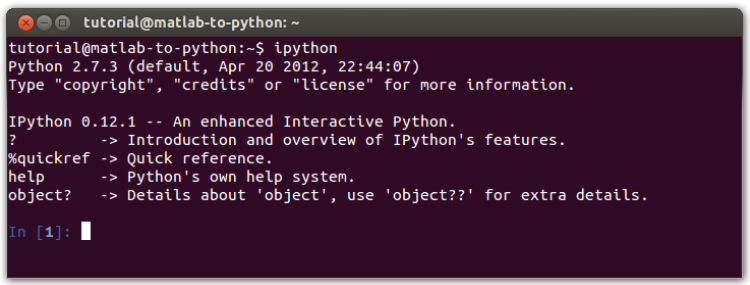
Introduction to Python

Playing with Python Interpreters

- IPython is a Python interpreter with additional interactive capabilities and more MATLAB-like features, discussed later.
- We will start with IPython to learn the basics and then move to Spyder when working with Python script files.
- Note: both Python and IPython can be run in a terminal with no need for a GUI.
  - To test this, click  in the launcher to open a terminal.
  - Type **python** to start Python and **ipython** to start IPython.
  - If your prompt is `>>>` you have a standard Python interpreter.
  - If your prompt is `In [1]:` you have an IPython interpreter.
  - Type **quit()** or **Ctrl-D** to exit, or close the terminal.



Python Interpreter



IPython Interpreter

Launch the IPython Qt Console

- Open the IPython Qt Console by clicking  in the launcher.

Basic Math

- Try typing commands directly into the IPython interpreter.

```
In [1]: 2 + 2          # This is a comment
Out[1]: 4

In [2]: 2.0 + 2.0      # Floating point addition
Out[2]: 4.0

In [3]: 5.0 / 2        # Floating point division
Out[3]: 2.5

In [4]: 5 / 2          # Integer division (in Python3 this returns 2.5)
Out[4]: 2

In [5]: 5.0 // 2.0     # Double slash forces integer division
Out[5]: 2.0

In [6]: 5 % 2          # 5 modulo 2, in MATLAB this is mod(5,2)
Out[6]: 1

In [7]: 3 ** 2         # Exponentiation, in MATLAB this is 3^2
Out[7]: 9

In [8]: _ + 2          # Underscore is the last returned value, in MATLAB this is 'ans'
Out[8]: 11
```

More Examples: `http://docs.python.org/tutorial/introduction.html`

Basic Built-in Types

- Basic Numerical Types
  - `int` - Integer type with unlimited precision (in Python 3).
  - `float` - Floating point type (usually double precision).
  - `complex` - Complex number with real and imaginary float components.
  - Note: in Python 2 `int` has limited precision and separate type `long` has unlimited precision.
- Special Types
  - `bool` - Boolean with values `False` or `True`.
  - `None` - A null object, more about this later.
- Container Types
  - `str` - A string of characters.
  - `tuple` - A fixed sequence of objects.
  - `list` - A mutable sequence of objects.
  - `dict` - A dictionary mapping one set of objects to another.
  - Plus various lesser used containers.
- Additional numerical types and containers are provided by NumPy.

More info: `http://docs.python.org/library/stdtypes.html`

Type Conversions

- Types are easily converted using the syntax `type(object)`
- Type conversion even works to and from strings
- Some type conversions (e.g. complex to float) are not well defined and throw an exception.

```
In [9]: float(7)
Out[9]: 7.0

In [10]: int(3.14159)
Out[10]: 3

In [11]: str(28)
Out[11]: '28'

In [12]: float('5.5')
Out[12]: 5.5

In [13]: complex(4.5)
Out[13]: (4.5+0j)

In [14]: bool(0)
Out[14]: False

In [15]: float(True)
Out[15]: 1.0
```

More info: `http://docs.python.org/library/stdtypes.html`

Assignment

- Variables are created during assignment

```
In [16]: my_var
---------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
NameError:  name 'my_var' is not defined

In [17]: my_var = 10

In [18]: my_var
Out[18]: 10
```

- Other assignment examples

```
In [19]: x, y, z = 10, 20, 30     # multiple assignment, uses tuples

In [20]: (y + z) / x
Out[20]: 5

In [21]: x += 5                   # eqivalent to x = x + 5

In [22]: y *= 2                   # eqivalent to x = x * 2

In [23]: x, y, z
Out[23]: (15, 40, 30)
```

More Examples: http://docs.python.org/tutorial/introduction.html

Complex Numbers

- Python natively supports complex numbers.
  - Imaginary numbers are written with the `j` or `J` suffix.

```
In [24]: 1j * 1J
Out[24]: (-1+0j)

In [25]: complex(3, 2)        # can also create with the 'complex' function
Out[25]: (3+2j)

In [26]: a = (-1 + 3j) * 2j

In [27]: a
Out[27]: (-6-2j)

In [28]: a.real               # access the real part
Out[28]: -6.0

In [29]: a.imag               # access the imaginary part
Out[29]: -2.0

In [30]: abs(a)               # abs(a) == sqrt(a.real**2 + a.imag**2)
Out[30]: 6.324555320336759

In [31]: a.conjugate()        # the complex conjugate
Out[31]: (-6+2j)
```

More Examples: `http://docs.python.org/tutorial/introduction.html`

Boolean Operators

Python	Result
`x or y`	if x is false, then y, else x (short-circuit)
`x and y`	if x is false, then x, else y (short-circuit)
`not x`	if x is false, then `True`, else `False`

MATLAB	Result
`x || y`	if `logical(x)`, then 1, else `logical(y)` (short-circuit)
`x && y`	if `logical(x)`, then `logical(y)`, else 0 (short-circuit)
`~x`	if `logical(x)`, then 0, else 1

- Examples

MATLAB

```
> 0.0 || 'hello'
1
> 10 && (5 || 6)
1
> ~ 5
0
```

Python

```
> 0.0 or 'hello'
'hello'
> 10 and (5 or 6)
5
> not 5
False
```

More info: `http://docs.python.org/library/stdtypes.html`

Comparisons

MATLAB	Python	Meaning
<	<	strictly less than
<=	<=	less than or equal
>	>	strictly greater than
>=	>=	greater than or equal
==	==	equal
~=	!=	not equal
N/A	`is`	object identity
N/A	`is not`	negated object identity

- These comparisons always return a Boolean value.
- Comparison between different numeric types casts from `int` to `float` to `complex` as needed. For example `3.0 == 3` is `True`.
- Only "not equal" operators differ in syntax.
- The concept of object identity is not found in MATLAB.

More info: `http://docs.python.org/library/stdtypes.html`

Object Identity

- Every Python object has an identity. The function `id(object)` returns a unique identifying integer.
- The operation `x is y` is equivalent to `id(x) == id(y)`.
- Assignment allows multiple variables to refer to the same object.
- For immutable types (e.g. numeric types, strings, tuples) creation of new objects may or may not refer to same-valued exiting object.
- For mutable type (e.g. list, dict) creation always creates a new object.

```
In [32]: x, y = 3, 4.4

In [33]: x == y, x is y, id(x), id(y)
Out[33]: (False, False, 30294792, 32375168)

In [34]: x = 2 + 2.4

In [35]: x == y, x is y, id(x), id(y)       # here 'x is y' could be True or False
Out[35]: (True, False, 37311776, 32375168)

In [36]: x = y

In [37]: x == y, x is y, id(x), id(y)       # here 'x is y' must be True
Out[37]: (True, True, 32375168, 32375168)
```

More info: http://docs.python.org/reference/datamodel.html

Tuples

- Tuples are immutable sequences of objects of any type.
- Tuple items are comma separated and often enclosed in parentheses.
- A trailing comma is required to make a tuple of length one.

```
In [38]: t = 1, 'hello', (3.14, None),  2.5

In [39]: t[0]          # access item by index
Out[39]: 1

In [40]: t[0] = 2      # item assignment not allowed (immutable)
TypeError:  'tuple' object does not support item assignment

In [41]: t[1:3]        # access a slice of the objects (start:end)
Out[41]: ('hello', (3.14, None))

In [42]: len(t)        # the length of the tuple
Out[42]: 4

In [43]: 2.5 in t      # test for inclusion
Out[43]: True

In [44]: t + (5,)      # concatenate tuples
Out[44]: (1, 'hello', (3.14, None), 2.5, 5)

In [45]: (1, 2) * 3    # concatenate n shallow copies
Out[45]: (1, 2, 1, 2, 1, 2)
```

More info: http://docs.python.org/library/stdtypes.html

Strings

- Strings are immutable sequences of characters.
- Strings are enclosed in single or double quotes.
- The concatenation and indexing syntax is the same as Tuple.
- Strings have many built-in functions.

```
In [46]: s = "Hello World"

In [47]: s.upper()                          # to upper case
Out[47]: 'HELLO WORLD'

In [48]: s[3:5] + '- ' * 3 + s[::-1]        # concatenation and slicing
Out[48]: 'lo- - - dlroW olleH'

In [49]: s.split()                          # string splitting
Out[49]: ['Hello', 'World']

In [50]: ' - '.join(('a', 'b', 'c'))        # join strings with a seperator
Out[50]: 'a - b - c'

In [51]: s.replace("Hello", "Goodbye")      # substring replacement
Out[51]: 'Goodbye World'

In [52]: "item %05d has value %g" % (23, 3.14**2)  # operator % works like sprintf
Out[52]: 'item 00023 has value 9.8596'
```

More info: http://docs.python.org/library/string.html

Lists

- Lists are mutable sequences of objects of any type.
- List items are comma separated and enclosed in square brackets.
- The concatenation and indexing syntax is the same as Tuple.

```
In [53]: v = [1, 'hello', (3.14, None),  2.5]

In [54]: v[0] = 2           # item assignment

In [55]: v.append(0.5)      # append items to the end

In [56]: del v[2]           # delete an item

In [57]: v
Out[57]: [2, 'hello', 2.5, 0.5]

In [58]: v.sort()           # sort items in place

In [59]: v
Out[59]: [0.5, 2, 2.5, 'hello']

In [60]: v = ([],)*3        # a tuple with 3 shallow copies of the same empty list

In [61]: v[0].append(1)     # append to the first list in the tuple

In [62]: v                  # all lists update because they are shallow copies
Out[62]: ([1], [1], [1])
```

More info: http://docs.python.org/library/stdtypes.html

Dictionaries

- Dictionaries are mutable mapping of one object to another.
- Key:value pairs are comma separated and enclosed in curly braces.

```
In [63]: d = {'a' : 1, 'b' : 2, 'c' : 3}

In [64]: d['a']        # access a value by key
Out[64]: 1

In [65]: d['d'] = 4    # change a value or insert a new key:value

In [66]: d.keys()      # a list of dictionary keys
Out[66]: ['a', 'c', 'b', 'd']

In [67]: d.values()    # a list of dictionary values
Out[67]: [1, 3, 2, 4]

In [68]: d.items()     # a list of (key, value) tuples
Out[68]: [('a', 1), ('c', 3), ('b', 2), ('d', 4)]
```

More info: `http://docs.python.org/library/stdtypes.html`

IPython Commands

- IPython adds several interactive capabilities to standard Python.
  - Tab Completion: start typing and hit the `Tab` key to get a list of possible completions. Works for variables, functions, paths, etc.
  - History: access previous commands and their results
    - adds `__` and `___` for second and third to last results.
    - use the command `history` to list the history of commands.
    - use `In[n]` to access the `n`-th command string.
    - use `Out[n]` to access the `n`-th result.
  - Documentation: type `objectName?` to get info about `objectName`. Note, the standard Python `help(objectName)` also provides some documentation.
  - Auto Call: lets you write functions as statements.
    For example, `abs x` calls `abs(x)`. MATLAB also allows this.
  - IPython provides many more features not covered here.

http://ipython.org/ipython-doc/rel-0.11/interactive/tutorial.html

IPython Commands

- Examples of using IPython history

```
In [1]: a = 1

In [2]: a += 1

In [3]: history
1: a = 1
2: a += 1
3: _ip.magic("history ")

In [4]: a
Out[4]: 2

In [5]: In[2]
Out[5]: u'a += 1\n'

In [6]: exec In[2]     # like 'eval' in MATLAB, Python also has 'eval' for expressions

In [7]: a
Out[7]: 3

In [8]: Out[4] + Out[7]
Out[8]: 5
```

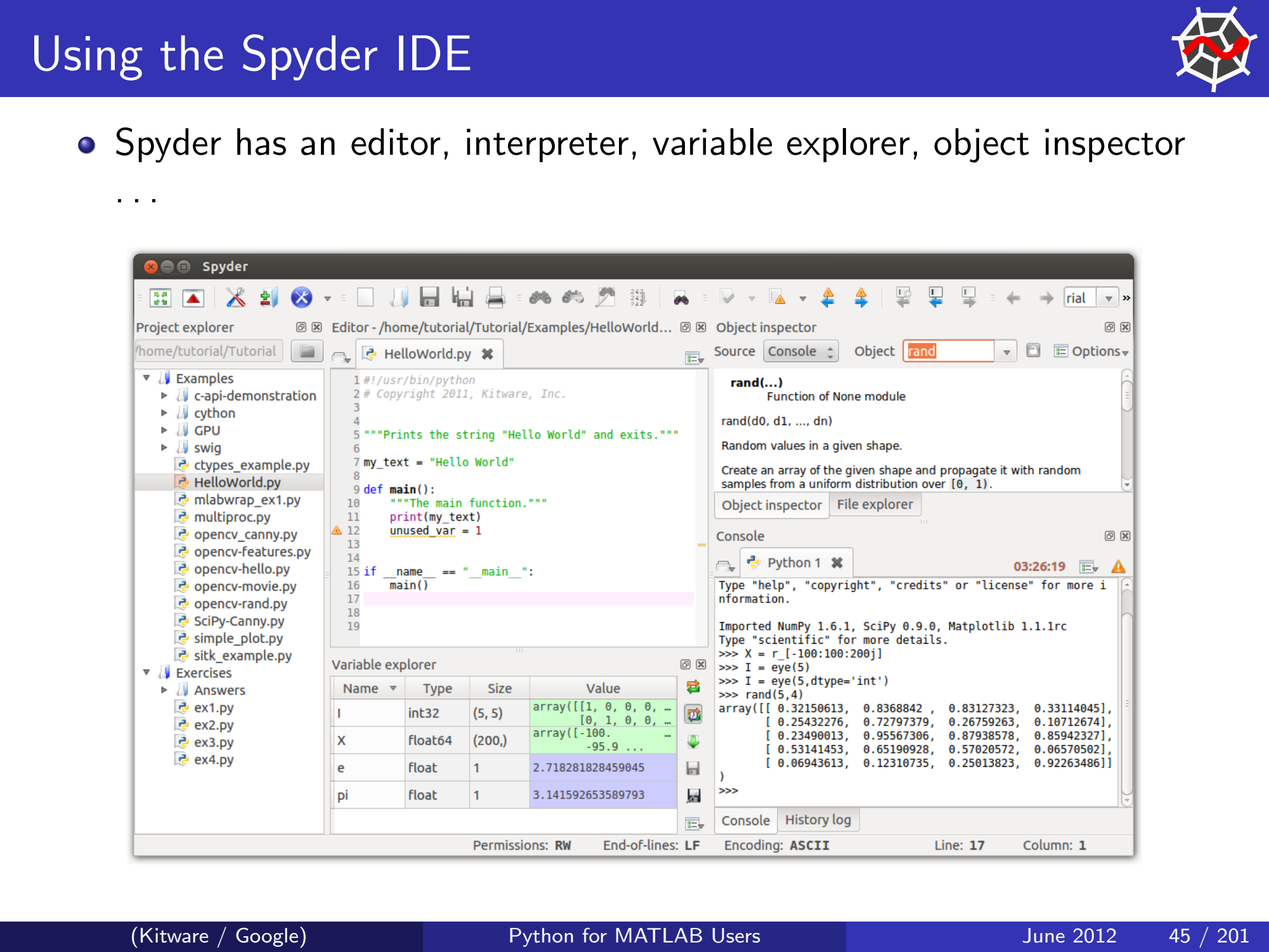
http://ipython.org/ipython-doc/rel-0.11/interactive/tutorial.html

Python Scripting

Launch the Spyder IDE

- So far we've been interacting with an interpreter.
  How do we write a script?
- Open the Spyder IDE by clicking  in the launcher.
- Note: This version of Spyder (2.1.9) does not fully integrate with the installed version of IPython (0.12).
  - The standard Python shell in Spyder has been enhanced with some features of IPython, like tab completion.
  - IPython will run within Spyder, but certain capabilities, like plotting, do not work.
  - IPython integration is scheduled to be fixed in Spyder 2.2, planned for October 2012.
  - For now we will use the standard interpreter, the one with the >>>prompt.

Using the Spyder IDE



- Spyder has an editor, interpreter, variable explorer, object inspector . . .

Hello World

- The simple way

```
print "Hello World"
```

- Note: In Python 3 `print` is a function, not a statement.

```
print("Hello World")  # Python 3 version
```

- A more complete example

Examples/HelloWorld.py

```
1  #!/usr/bin/python
2  # Copyright 2011, Kitware, Inc.
3
4
5  """Prints the string "Hello World" and exits."""
6
7  my_text = "Hello World"
8
9  def main():
10     """The main function."""
11     print my_text
12
13
14 if __name__ == "__main__":
15     main()
```

Hello World: line-by-line

- Comments. The first line is a shebang directive for Unix.

```
1  #!/usr/bin/python
2  # Copyright 2011, Kitware, Inc.
```

- Documentation string (docstring) for the script. Uses triple quotes.

```
5  """Prints the string "Hello World" and exits."""
```

- Assigns the "Hello World" string to a global variable.

```
7  my_text = "Hello World"
```

- A function definition with docstring. It calls the print function.

```
9  def main():
10     """The main function."""
11     print my_text
```

- Call the main function only if this file is run as a script (not imported).

```
14 if __name__ == "__main__":
15     main()
```

Running a Script

- In Spyder run the active script with Menu "Run→Run", or press F5
- Adjust the configuration settings and press the "Run" button
- Future runs use the same configuration
- Change the run configuration with Menu "Run→Configure", or press F6

- Alternatively, in a terminal window (from the home directory) run `python Tutorial/Examples/HelloWorld.py`

Conditionals

MATLAB

```
if a > 10
    a = 10;
end
```

Python

```
if a > 10:
    a = 10
```

- Syntax of basic conditionals is very similar.
- MATLAB uses keyword `end` to designate the end of a block.
- Python uses : to start a block and indentation to designate duration.
  - These rules apply generally to loops, functions, classes, etc.
  - **Indentation is important in Python!**
  - Choose any indentation, but consistent indentation defines a block.

MATLAB

```
if a > 10
    disp('a greater than 10')
elseif a < 5
    disp('a less than 5')
else
    disp('a is ', a)
end
```

Python

```
if a > 10:
    print 'a greater than 10'
elif a < 5:
    print 'a less than 5'
else:
    print 'a is', a
```

- Note, `elseif` in MATLAB becomes `elif` in Python.

Loops

MATLAB

```
x = 1;
while x < 10
    x = x * 2;
end
```

Python

```
x = 1
while x < 10:
    x = x * 2
```

- MATLAB and Python `while` loops are essentially the same.
- `break` and `continue` statements are the same in both languages.

MATLAB

```
X = [5 4 3 2 1];
for v = X
    disp(v)
end
```

Python

```
X = [5, 4, 3, 2, 1]
for v in X:
    print v
```

- MATLAB and Python `for` loops each iterate over a container.
- MATLAB uses symbol = while Python uses keyword `in`.

Generating Ranges

- MATLAB has special syntax to generate a range of numbers
  - `first:last`
  - `first:step:last`
- Python use the `range` function to achieve similar behavior
  - `range(end)`
  - `range(first, end)`
  - `range(first, end, step)`

MATLAB

```
> 0:4
 0   1   2   3   4
> 2:5
 2   3   4   5
> 6:-2:2
 6   4   2
```

Python

```
> range(5)
[0, 1, 2, 3, 4]
> range(2, 6)
[2, 3, 4, 5]
> range(6, 1, -2)
[6, 4, 2]
```

- Note that `last` is inclusive while `end` is not. In MATLAB `first`$\leq i \leq$`last`, but in Python `first`$\leq i <$`end`.
- Python's `range` function is limited to integer types. NumPy adds the `arange` function which generalizes to floats.

Defining Functions

MATLAB

```
function [s, d] = sum_diff(x, y)
% SUM_DIFF Sum and diff of two values.
% x is the first value
% y is the second value

s = x + y;
d = x - y;
```

Python

```
def sum_diff(x, y):
    """Sum and diff of two values.

    x is the first value
    y is the second value
    """
    return x + y, x - y
```

- MATLAB handles return values by explicitly naming them.
  - returns zero, one, or multiple named variables.
- Python always returns exactly one object with the `return` statement.
  - a standalone or missing `return` statement returns `None`.
  - returning multiple comma-separated objects (as above) returns a tuple.

MATLAB

```
> [v1, v2] = sum_diff(7, 4)
v1 = 11
v2 = 3
> v3 = sum_diff(7, 4)
v3 = 11
```

Python

```
> v1, v2 = sum_diff(7, 4)
> v3 = sum_diff(7, 4)
> print "v1 = ", v1, ", v2 = ", v2, \
        ", v3 = ", v3
v1 = 11, v2 = 3, v3 = (11, 3)
```

Defining Functions – Optional Arguments

- MATLAB function arguments are all optional, but you can enforce the number of arguments and set default values by checking `nargin`.
- Python function arguments are required unless a default value is specified.
- All optional arguments must come after all required arguments.

MATLAB

```
function val = my_func(req, opt1, opt2)
if nargin < 3
    opt2 = 'default string';
elseif nargin < 2
    opt1 = 3.14;
end
disp(opt2)
val = req * opt1
```

Python

```
def my_func(req, opt1=3.14,
            opt2='default string'):
    print opt2
    return req * opt1
```

- Python optional arguments can be omitted and/or specified as `key=value` pairs in any order, e.g in the above example `my_func(1, 3.14, 'hi')` is the same as `my_func(1, opt2='hi')`

http://docs.python.org/tutorial/controlflow.html#keyword-arguments

Managing Libraries of Functions

- MATLAB and Python have different strategies for organizing and calling libraries of functions.
- In MATLAB, importing functions is implicit.
  - Each function must reside in a file of the same name.
  - When a function is called, the MATLAB path is searched to find the first matching function.
  - Dealing with functions in different packages with the same name is problematic.
- In Python, importing functions is explicit.
  - Any number of functions can reside in a file.
  - Modules, packages, and function must be explicitly imported when needed.
  - Name spaces and renaming allows for easy resolution of name conflicts.

Importing Packages, Modules, Functions, etc.

- The Python `import` statement is used to explicitly import packages, modules, classes, functions, and variables.

  module is a python file (`.py`) that may contain multiple functions, variables, and classes.

  package is a collection of modules in a directory. The directory must contain a file named `__init__.py`. Packages can be nested.

- The `import` statement initializes a module or package by executing the code in the module or `__init__.py`.
- Python searches current directory and Python path to find matching modules and packages.
- Each module or package is imported **only once** in a Python session, call `reload()` to force reloading.

http://docs.python.org/tutorial/modules.html

An Example Python Package

Directory structure of a Python package

```
vision/                          Top-level package
      __init__.py                Initialize vision package
      learning/                  Subpackage for learning
               __init__.py
               adaboost.py
               svm.py
      tracking/                  Subpackage for tracking
               __init__.py
               kalman.py
      features/                  Subpackage for features
               __init__.py
               sift.py
               harris.py
               canny.py
```

http://docs.python.org/tutorial/modules.html

Importing Modules and Packages

- If the `vision` directory is in the Python path then the following works.

```
import vision.tracking.kalman           # executes vision/tracking/kalman.py
vision.tracking.kalman.predict()        # call a function in kalman.py
```

- Generally `__init__.py` includes statements to import all the sub-packages or modules recursively. So the following usually works, but it imports the whole vision package.

```
import vision                           # executes vision/__init__.py
vision.features.harris.detect()         # call a function in harris.py
vision.tracking.kalman.predict()        # call a function in kalman.py
```

- The `as` keyword allows renaming during import.

```
import vision.tracking.kalman as kf
kf.predict()
```

http://docs.python.org/tutorial/modules.html

Importing Specific Functions or Variables

- The `from` keyword allows importing into the current name space.

```
from vision.tracking.kalman import predict, correct
predict()
correct()
```

- Rename specific functions with `as`.

```
from vision.tracking.kalman import predict as my_pred, correct as my_corr
my_pred()
my_corr()
```

- Using * imports of everything into the current name space.

```
from vision.tracking.kalman import *
predict()
correct()
```

http://docs.python.org/tutorial/modules.html

The Python Path

- The Python path is set with environment variable `PYTHONPATH`.
- Modify `sys.path` to make temporary changes during execution.

```
In [9]: import sys

In [10]: sys.path.append("/home/tutorial")

In [11]: sys.path
Out[11]:
['',
 '/usr/bin',
 '/usr/lib/python2.7',
 ...
 '/usr/lib/python2.7/dist-packages/IPython/extensions',
 '/home/tutorial']
```

http://docs.python.org/tutorial/modules.html

Exercise 1

Basic Python

- Open the file `Exercises/ex1.py`
- Write the missing function `sqrt_dict` to compute a dictionary mapping the first `n` integers to the square roots of their values.
- Hint: use the `sqrt` function from the `math` module.
- This example uses `sys.argv` to access a list of command line arguments.

Exercises/ex1.py

```
19      # parse the command line argument
20      # require exactly 1 argument (argv[0] is the script name)
21      if len(sys.argv) != 2:
22          print "usage: %s number" % sys.argv[0]
23          return
24
25      # convert the command line argument to an interger
26      num = int(sys.argv[1])
27      # call the function
28      values = sqrt_dict(num)
```

Exercise 1: Answer

Basic Python

- Here is one possible answer

Exercises/Answers/ex1-answer.py

```
6  import sys
7  import math
8
9
10 def sqrt_dict(n):
11     """Compute a dictionary mapping the first n integers to square roots."""
12     d = dict()
13     for i in range(n):
14         d[i] = math.sqrt(i)
15     return d
```

- An alternate one-line solution uses list comprehensions

Exercises/Answers/ex1-answer.py

```
22     return dict([(i, math.sqrt(i)) for i in range(n)])
```

Scientific Computing in Python

NumPy, SciPy, and Matplotlib

- NumPy adds basic MATLAB-like capability to Python:
  - multidimensional arrays with homogeneous data types
  - specific numeric data types (e.g. `int8`, `uint32`, `float64`)
  - array manipulation functions (e.g. reshape, transpose, concatenate)
  - array generation (e.g. ones, zeros, eye, random)
  - element-wise math operations (e.g. add, multiply, max, sin)
  - matrix math operations (e.g. inner/outer product, rank, trace)
  - linear algebra (e.g. inv, pinv, svd, eig, det, qr)
- SciPy builds on NumPy (much like MATLAB toolboxes) adding:
  - multidimensional image processing
  - non-linear solvers, optimization, root finding
  - signal processing, fast Fourier transforms
  - numerical integration, interpolation, statistical functions
  - sparse matrices, sparse solvers
  - clustering algorithms, distance metrics, spatial data structures
  - file IO (including to MATLAB .mat files)
- Matplotlib adds MATLAB-like plotting capability on top of NumPy.

Interactive Scientific Python (aka PyLab)

- PyLab is a meta-package that includes NumPy, SciPy, and Matplotlib.
- The easiest (and most MATLAB-like) way to work with scientific Python is to import everything into the global name space.

```
from pylab import *
```

- IPython can be started with the `--pylab` command line option.
- Spyder starts its standard Python interpreter with these scientific libraries imported, much like PyLab, but not PyLab *per se*.

More info: `http://www.scipy.org/PyLab`

Scripting with Scientific Python

- When writing scripts it is recommended that you:
  - only import what you need, for efficiency
  - import packages into namespaces, to avoid name clashes
- The community has adopted abbreviated naming conventions:

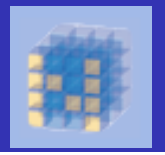
```
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
```

- Some different ways of working with NumPy are

```
from pylab import *             # Import everything.  Good for interactive work,
svd(eye(3))                     # but not recommended for writing scripts.

from numpy import eye, array    # Import only what you need into the global namespace.
from numpy.linalg import svd    # Verbose syntax for import statements,
svd(eye(3))                     # but simple syntax for calling functions.

import numpy as np              # Import a superset of what you need into a namespace.
np.linalg.svd(np.eye(3))        # More verbose syntax for calling functions.
```

http://docs.scipy.org/doc/scipy/reference/tutorial/general.html

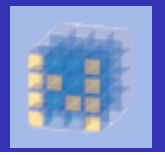
NumPy



- Numpy provides a good website for mapping MATLAB syntax into equivalent NumPy syntax:
  `http://www.scipy.org/NumPy_for_Matlab_Users`
- This tutorial will only cover the most common cases.

More info: `http://www.scipy.org/NumPy_for_Matlab_Users`

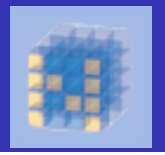
NumPy Array



- Use `numpy.array` to represent multidimensional arrays.

```
>>> A = array([1, 2, 3])    # create a one dimensional array from a list

>>> B = array([[1, 2, 3],
               [4, 5, 6]])  # create a two dimensional array from a list of lists

>>> C = array([B, B, B, B]) # create a 3D array from a list of 2D arrays

>>> A.shape                 # the size of each dimension
(3,)

>>> B.shape
(2,3)

>>> C.shape
(4,2,3)

>>> A.ndim                  # the number of dimensions
1

>>> A.dtype                 # the data type of each element
dtype('int64')
```

NumPy Array



- Unlike MATLAB, operations on `numpy.arrays` are element-wise.

MATLAB

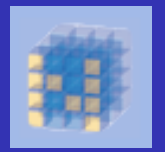
```
> M = [1 2; 2 1];
> disp(M*M)
    5    4
    4    5

> disp(M.*M)
    1    4
    4    1
```

Python

```
> M = array([[1, 2],[2, 1]])
> print M*M
[[1 4]
 [4 1]]

> print dot(M,M)
[[5 4]
 [4 5]]
```

- The function `numpy.dot` is required for matrix (or tensor) multiplication.
- This syntax may seem ugly, but makes operations more consistent over arrays with dimensions greater than 2.

NumPy Matrix



- Numpy provides a special `numpy.matrix` subclass of `numpy.array`.
- `numpy.matrix` acts more like a MATLAB matrix.

MATLAB

```
> M = [1 2; 2 1];
> disp(M*M)
   5   4
   4   5
```

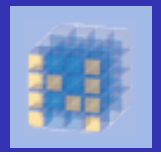
Python

```
> M = matrix('1 2; 2 1')
> print M*M
[[5 4]
 [4 5]]
```

- `numpy.matrix` can use a MATLAB-style string initializer.
- Use "`from numpy.matlib import *`" to override common array constructors (`zeros()`, `eye()`, etc.) with matrix versions.
- `numpy.matrix` only handles two dimensional arrays.
- `numpy.array` is more consistent and still preferred in most cases.

More info: `http://www.scipy.org/NumPy_for_Matlab_Users`

NumPy Numerical Data Types



- Numpy adds several numerical data types to Python

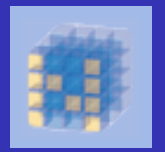
Names	Descriptions
`bool`	Boolean (`True` or `False`) stored in a byte
`int`	Platform integer (usually `int32` or `int64`)
`int8`, `int16`, `int32`, `int64`	8, 16, 32 and 64 bit signed integers
`uint8`, `uint16`, `uint32`, `uint64`	8, 16, 32 and 64 bit unsigned integers
`float`	Shorthand for `float64`
`float16`, `float32`, `float64`	half, single, and double precision floats
`complex`	Shorthand for `complex128`
`complex64`, `complex128`	complex numbers with single and double precision floats for real and imaginary parts

- Type can be specified during array creation and converted later

```
>>> A = np.array([1, 2, 3], dtype=np.float32)   # construct as type float32

>>> A
array([ 1.,  2.,  3.], dtype=float32)

>>> A.astype('complex')                # convert the type of an existing array
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

More info: http://docs.scipy.org/doc/numpy/user/basics.types.html

NumPy Array Construction



- Common helper functions for constructing arrays are similar to MATLAB.

MATLAB

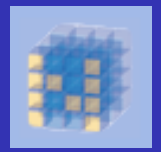
```
> M = zeros(3,4);
> M = ones(3,4);
> M = eye(3);
> M = diag([1 2 3]);
> M = diag(A);
> M = rand(3,4);
> M = repmat(A, m, n);
```

Python

```
> M = zeros((3,4))
> M = ones((3,4))
> M = eye(3)
> M = diag([1, 2, 3])
> M = diag(A)
> M = random.rand(3,4)
> M = tile(A, (m, n))
```

More info: http://www.scipy.org/NumPy_for_Matlab_Users

NumPy Array Slicing

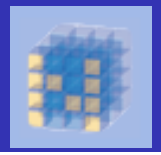


- Slicing allow access to subset of arrays.
- Python slicing syntax is `first:end:step`.

MATLAB	NumPy	Description
`a(end)`	`a[-1]`	access last element in the 1xn matrix `a`.
`a(2,5)`	`a[1,4]`	access element in second row, fifth column.
`a(2,:)`	`a[1]` or `a[1,:]`	access entire second row of `a`.
`a(1:5,:)`	`a[0:5]` or `a[:5]` or `a[0:5,:]`	access the first five rows of `a`.
`a(3:2:21,:)`	`a[ 2:21:2,:]`	every other row of `a`, from the third to the twenty-first.
`a(1:2:end,:)`	`a[ ::2,:]`	every other row of `a`, starting with the first.
`a(end:-1:1,:)`	`a[ ::-1,:]`	`a` with rows in reverse order.

More info: `http://www.scipy.org/NumPy_for_Matlab_Users`

NumPy Indexing Tricks – Stacking



- Numpy provides a special class to enable several indexing tricks.
- The `r_` class provides row stacking of arrays
- `r_` is used with brackets as if it is being indexed.

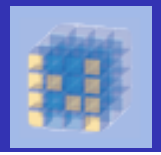
```
>>> r_[[1, 2, 3], [4, 5, 6], [7, 8]]   # stack 1D arrays along the first axis
 array([1, 2, 3, 4, 5, 6, 7])

>>> r_[eye(2), 2*ones((2,2)) ]         # stack 2D arrays along the first axis

array([[ 1.,  0.],
       [ 0.,  1.],
       [ 2.,  2.],
       [ 2.,  2.]])
```

- A variant named `c_` is same, but for column stacking.

```
>>> c_[eye(2), 2*ones((2,2)) ]         # stack 2D arrays along the second axis

array([[ 1.,  0.,  2.,  2.],
       [ 0.,  1.,  2.,  2.]])
```

More info: http://www.scipy.org/NumPy_for_Matlab_Users

NumPy Indexing Tricks – Range Generation



- The `r_` class also provides range generation.
- If an argument uses slice notation it is expanded to a range
- `r_[first:end:step]` is the same as `arange(first,end,step)`

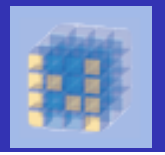
```
>>> r_[5:10:0.5]      # generates a range from 5 to 10 with step 0.5
array([ 5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
```

- When the step is replaced with an imaginary number the slice has a different interpretation, `first:last:num`

```
>>> r_[5:10:5j]      # generates 5 samples between 5 and 10, inclusive
array([  5.  ,   6.25,   7.5 ,   8.75,  10.  ])
```

More info: `http://www.scipy.org/NumPy_for_Matlab_Users`

NumPy Indexing Tricks

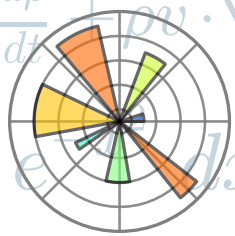


- The `ix_` function generates indexes for irregular slices

MATLAB	NumPy	Description
`a([2,4,5],[1,3])`	`a[ix_([1,3,4],[0,2])]`	rows 2,4 and 5 and columns 1 and 3.

More info: `http://www.scipy.org/NumPy_for_Matlab_Users`

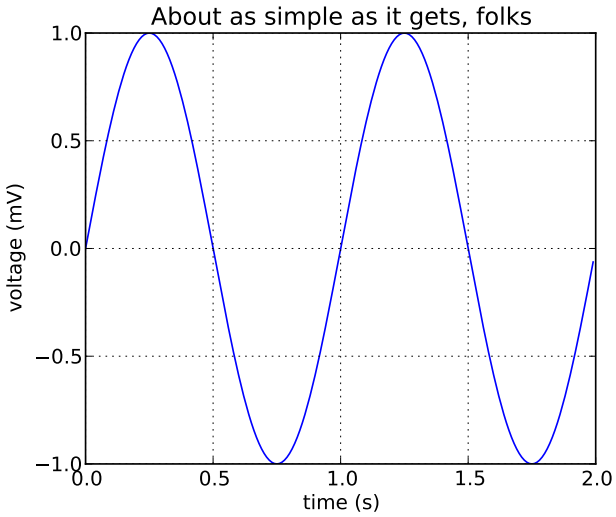
Matplotlib

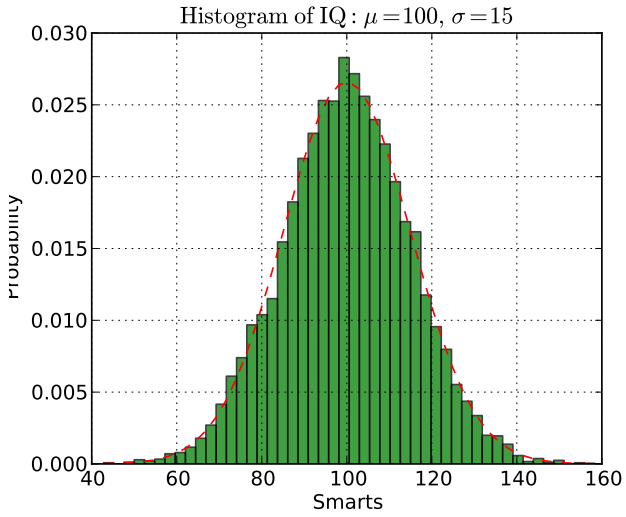


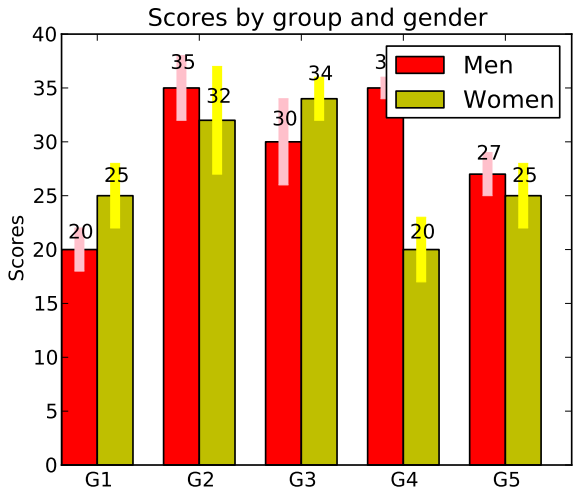
- Matplotlib provides plotting capabilities that are similar to MATLAB.
- Syntax is the same or similar to MATLAB in most cases.
- Matplotlib has some advantages over MATLAB plots
  - Handles TeX equation typesetting within plots.
  - Easy export to PDF using vector graphics.
  - Plots often look nicer than MATLAB plots.
- The above logo is actually a matplotlib plot.

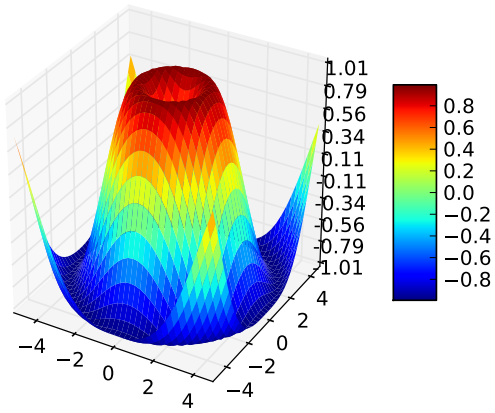
More info: `http://matplotlib.sourceforge.net/`

Matplotlib Example Plots







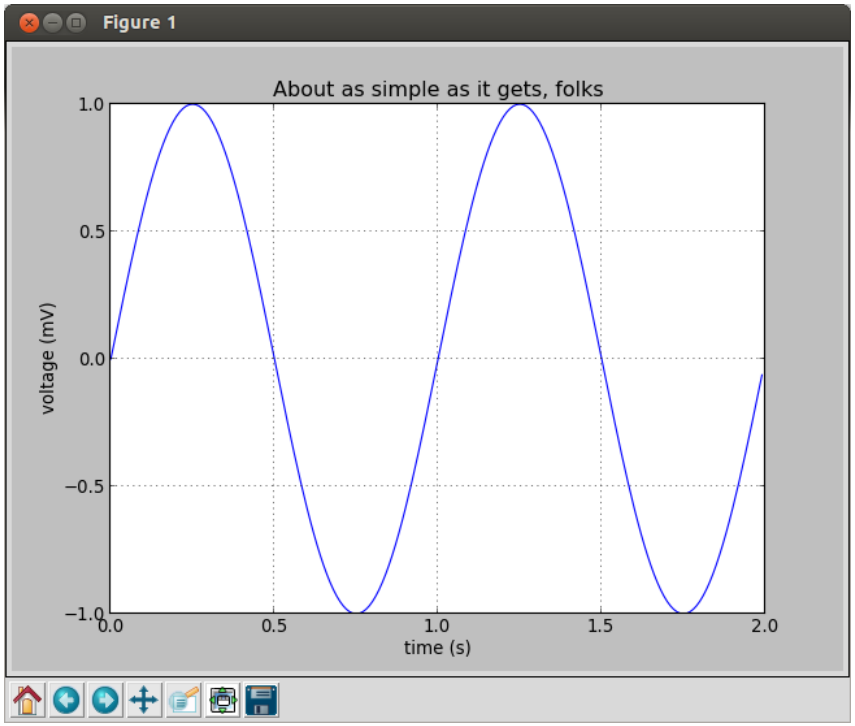


More info: `http://matplotlib.sourceforge.net/users/screenshots.html`

Matplotlib Example Plots

Examples/simple_plot.py

```
1  from pylab import *
2
3  t = arange(0.0, 2.0, 0.01)
4  s = sin(2 * pi * t)
5  plot(t, s, linewidth=1.0)
6
7  xlabel('time (s)')
8  ylabel('voltage (mV)')
9  title('About as simple as it gets, folks')
10 grid(True)
11 show()
```



- When not using pylab, these functions are found in `matplotlib.pyplot`.
- The figure is not displayed until `show()` is called.

More info: `http://matplotlib.sourceforge.net/users/screenshots.html`

Matplotlib Interactive Plotting

- Matplotlib has an interactive plotting mode that updates plots after each command, and `show()` is not needed.
- Interactive plotting is enabled with `ion()` and disabled with `ioff()`.
- Interactive mode is enabled automatically in Spyder and `ipython --pylab`

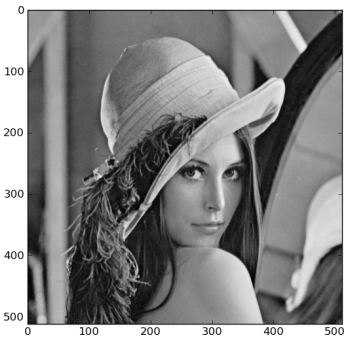
More info: `http://matplotlib.sourceforge.net/users/shell.html`

Exercise 2

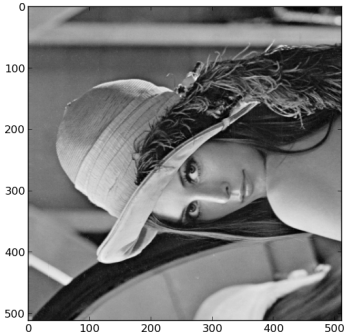
Plotting Lena with NumPy, SciPy, and Matplotlib

```
>>> from scipy.misc import lena
>>> imshow(lena())
>>> gray()
```

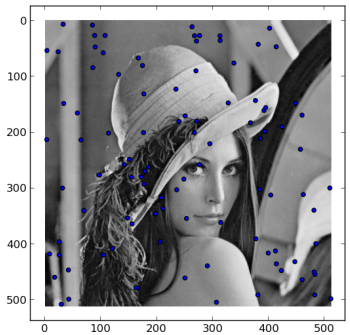
1. Plot the "Lena" image.
2. Plot the transpose of Lena using `lena().T`.
3. Plot 100 random points on top of Lena with x and y between 0 and 512 using `rand` and `scatter`.
4. Plot a histogram of Lena with 100 bins using `hist`. Hint: Flatten the image to 1-D using `lena().flatten()`
5. Plot Lena with a sub-matrix from (200,200) to (400,400) flipped horizontally. Hint: Create a modified image array using slice notation.



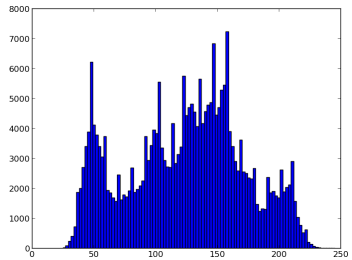
Part 1



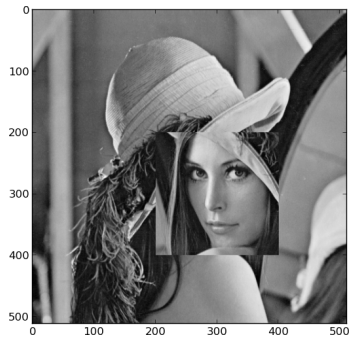
Part 2



Part 3



Part 4



Part 5

Exercise 2: Answers

Plotting Lena with NumPy, SciPy, and Matplotlib

Exercises/Answers/ex2-answer.py

```
7  from scipy.misc import lena
8
9  figure("Part 1")
10 imshow(lena())
11 gray()
12
13 figure("Part 2")
14 imshow(lena().T)
15 gray()
16
17 figure("Part 3")
18 imshow(lena())
19 points = rand(2, 100) * 512
20 scatter(points[0], points[1])
21
22 figure("Part 4")
23 hist(lena().flatten(), 100)
24
25 figure("Part 5")
26 image = lena()
27 image[200:400, 200:400] = image[200:400, 400:200:-1]
28 imshow(image)
```

SciPy

- Contains many scientific Python modules built on top of NumPy.
- Most interesting for computer vision work are

ndimage	Multi-dimensional image processing
linalg	Linear Algebra (extends numpy.linalg)
fftpack	Fourier Transforms
optimize	Optimization
io	File IO (read/write MATLAB files)

- In this tutorial, we are only going to scratch the surface of what is contained in these libraries with a few examples.

More info: `http://www.scipy.org/`

SciKits

- There are also many add-on toolkits for SciPy, collectively called SciKits
- Most interesting for computer vision work are

scikits-image	Image Processing
scikits-learn	Machine Learning and Data Mining
ann	Approximate Nearest Neighbor library wrapper
mlabwrap	Call MATLAB from Python

More info: `http://scikits.appspot.com/`

SciPy Central

- Looking for an equivalent to MATLAB Central for sharing snippets of scientific Python code?
- Try SciPy Central (scipy-central.org)
- It's not as extensive as MATLAB Central, but it's fairly new.

More info: `http://scipy-central.org/`

SciPy By Example

- Let's work through an example of writting a Canny edge detector.
- This is not a very efficient implementation of Canny edge detection.
- The goal is to illustrates some SciPy and Numpy functionality.
- Open the file Examples/scipy_canny.py

SciPy By Example – Importing

- First import what we need: NumPy and and NDImage from SciPy.
- These modules are imported to abreviated names `np` and `ndi`.
- Note that the constant `pi` (i.e. π) is imported into the global namespace to make math expressions easier to follow.

Examples/scipy_canny.py

```
4  import numpy as np
5  from numpy import pi
6  import scipy.ndimage as ndi
```

SciPy By Example – Main

- Now look at the end of the file.

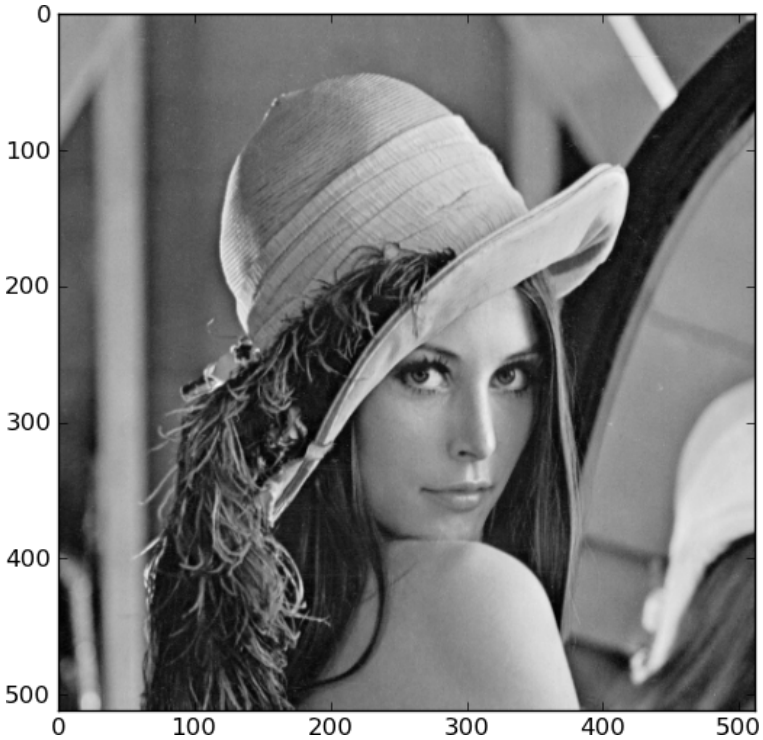
Examples/scipy_canny.py

```
46  if __name__ == "__main__":
47      from scipy.misc import lena
48      from matplotlib.pyplot import imshow, gray, show
49      imshow(canny_edges(lena()))
50      gray()
51      show()
```

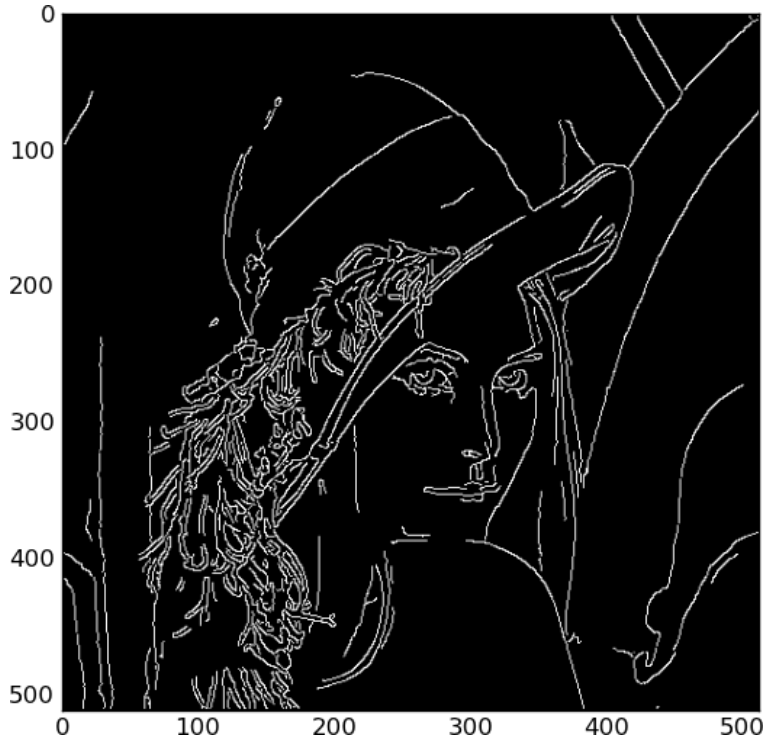
- Usually all imports happen at the top of a file, but this is not required.
- In this example some more things are imported only when this file is run as a script.
- When run as a script, this code calls `canny_edges` on the Lena image and displays the result in a matplotlib GUI.
- When imported into another module, there is no dependency on matplotlib or Lena.

SciPy By Example – Result

- Calling `imshow(lena())` would show the input image on the left.
- Running this script displays the image on the right.



Input Image



Canny Edges

SciPy By Example – Canny

- Here is the Canny edge detection code.

Examples/scipy_canny.py

```
27  def canny_edges(image, sigma=1.0, low_thresh=50, high_thresh=100):
28      """Compute Canny edge detection on an image."""
29      image = ndi.filters.gaussian_filter(image, sigma)
30      dx = ndi.filters.sobel(image, 0)
31      dy = ndi.filters.sobel(image, 1)
32
33      mag = np.sqrt(dx ** 2 + dy ** 2)
34      ort = np.arctan2(dy, dx)
35
36      edge_map = non_maximal_edge_suppresion(mag, ort)
37      edge_map = np.logical_and(edge_map, mag > low_thresh)
38
39      labels, num_labels = ndi.measurements.label(edge_map, np.ones((3, 3)))
40      for i in range(num_labels):
41          if max(mag[labels == i]) < high_thresh:
42              edge_map[labels == i] = False
43      return edge_map
```

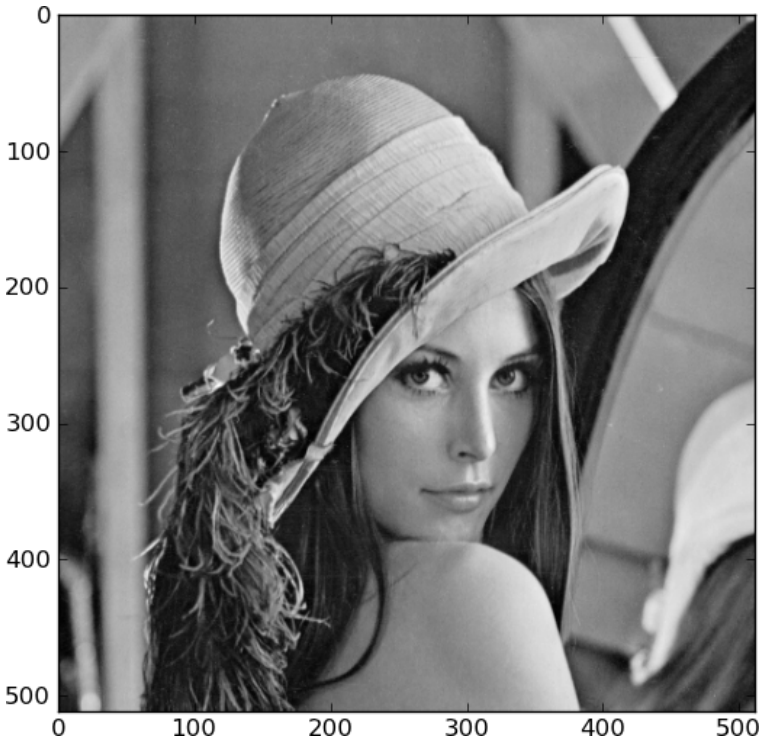
- Only `non_maximal_edge_suppression` is not provided by `np` or `ndi`.

SciPy By Example – Gaussian Filter

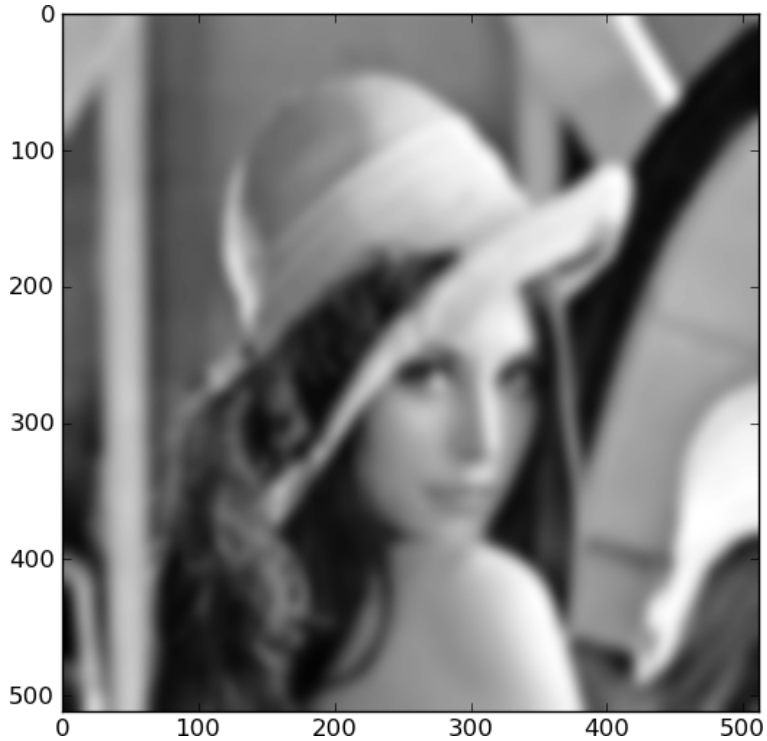
1. Smooth the image with a Gaussian Filter.

Examples/scipy_canny.py

```
29      image = ndi.filters.gaussian_filter(image, sigma)
```



Input Image

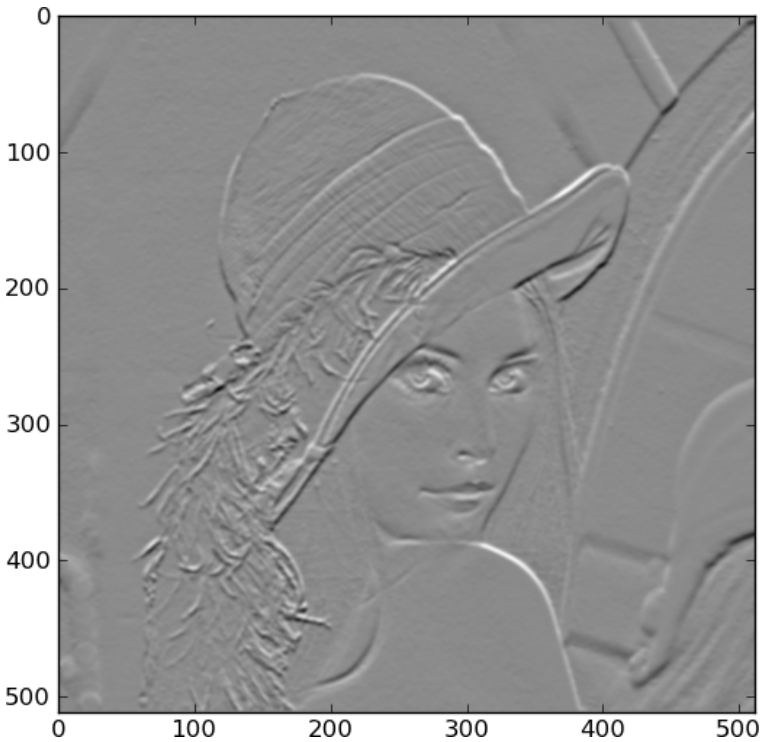


Gaussian Filter with $\sigma = 5$

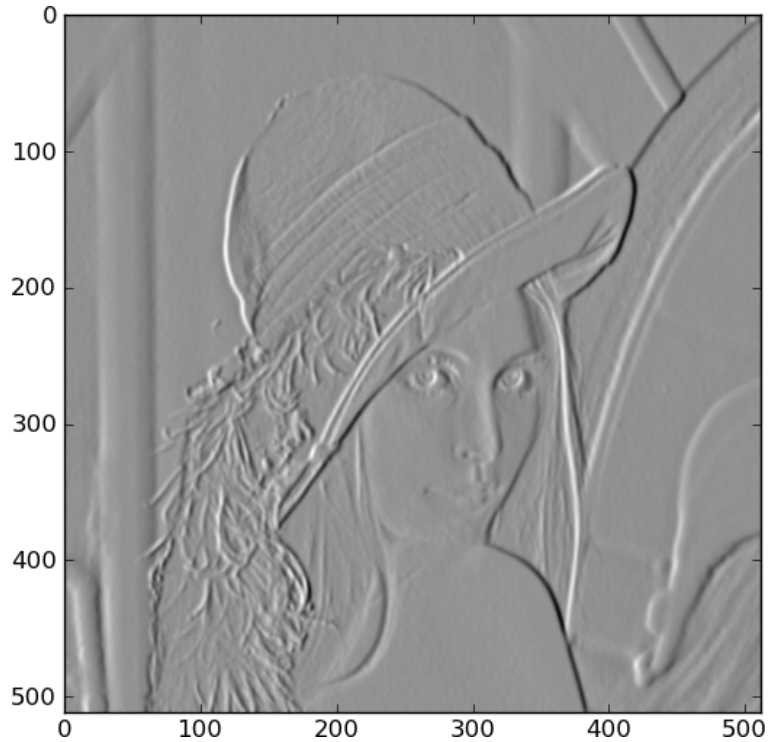
2. Sobel derivatives on smoothed image.

Examples/scipy_canny.py

```
30      dx = ndi.filters.sobel(image, 0)
31      dy = ndi.filters.sobel(image, 1)
```



dx – Sobel horizontal

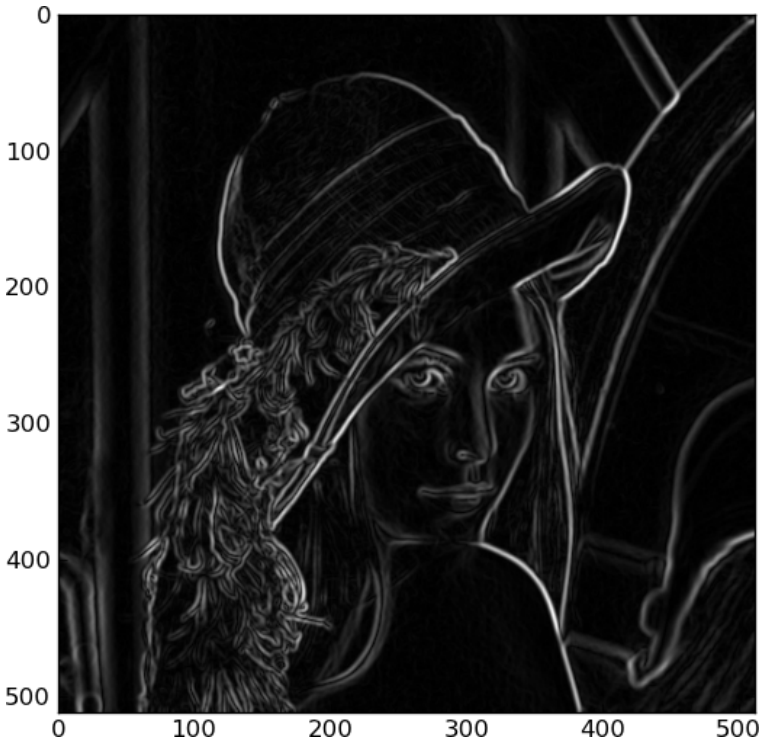


dy – Sobel vertical

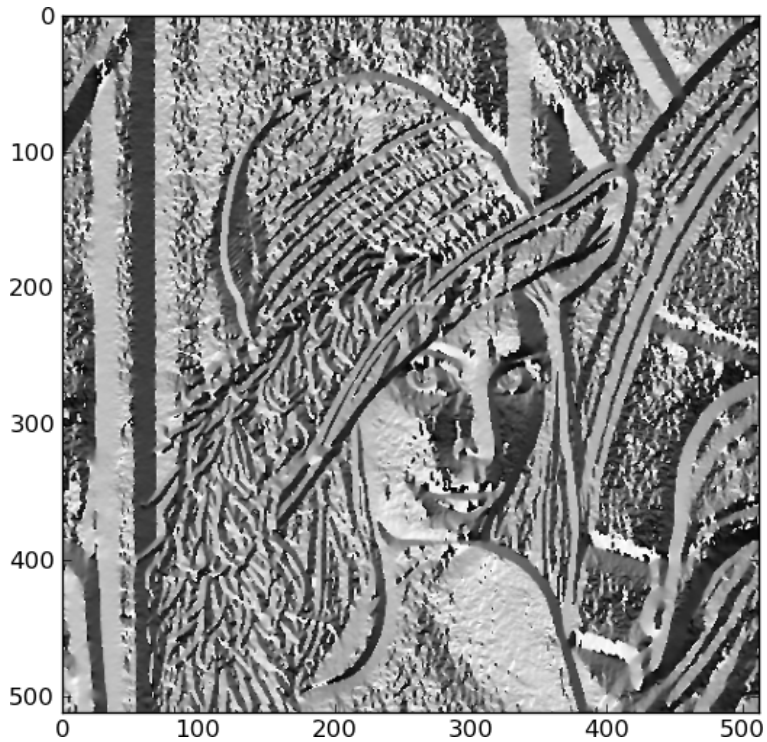
3. Sobel derivatives on smoothed image.

Examples/scipy_canny.py

```
33      mag = np.sqrt(dx ** 2 + dy ** 2)
34      ort = np.arctan2(dy, dx)
```



Gradient Magnitude



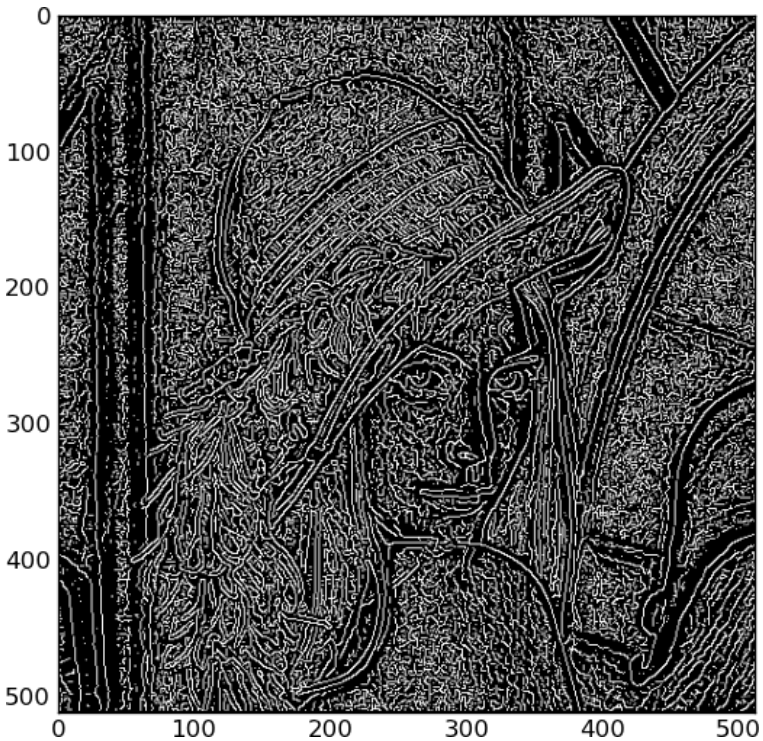
Gradient Orientation

SciPy By Example – Non-Max Suppression

4. Non-maximal suppression (more on this later).

Examples/scipy_canny.py

```
36      edge_map = non_maximal_edge_suppresion(mag, ort)
```

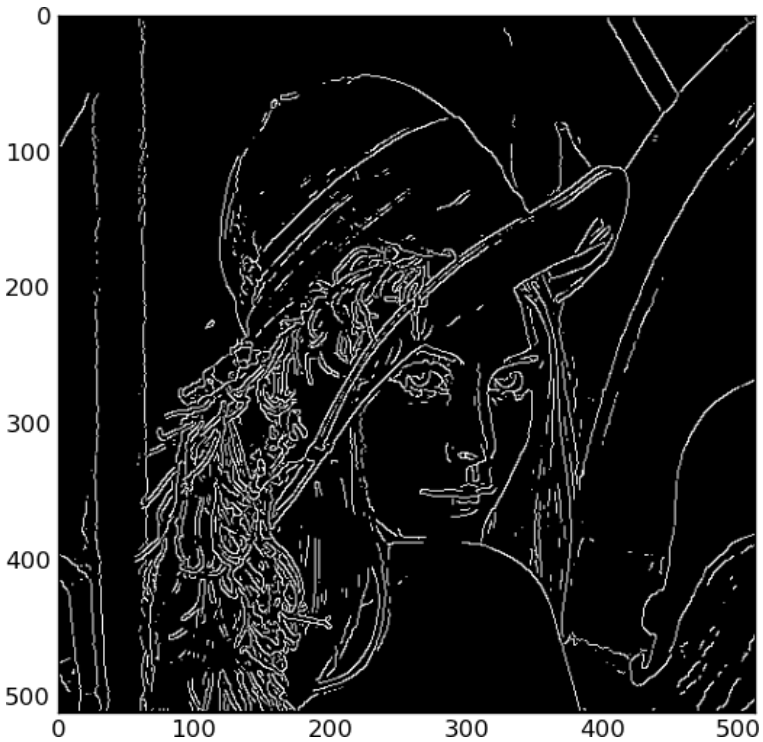


All maximimal edges (binary image)

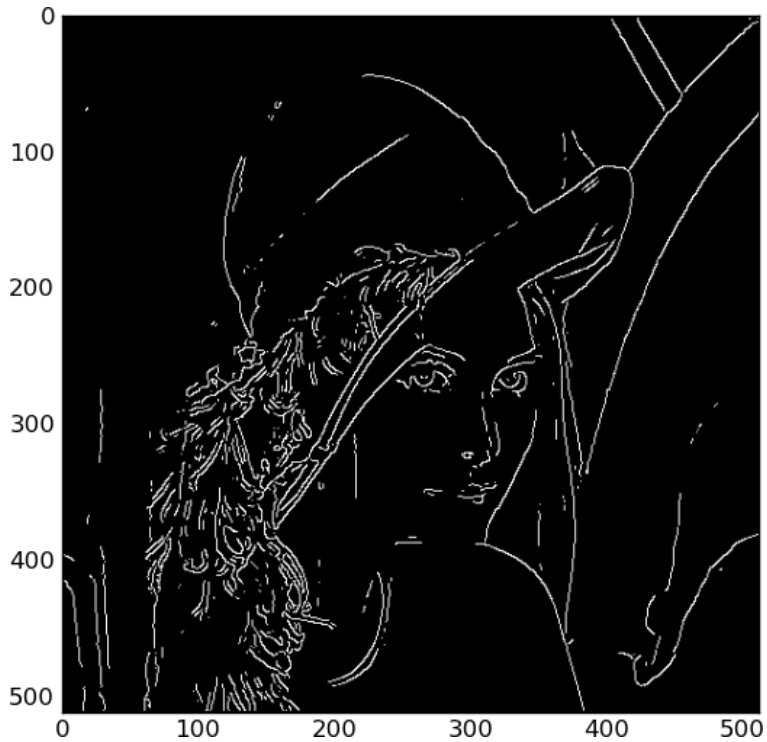
5. Apply the lower threshold

Examples/scipy_canny.py

```
37      edge_map = np.logical_and(edge_map, mag > low_thresh)
```



Low Threshold (50)



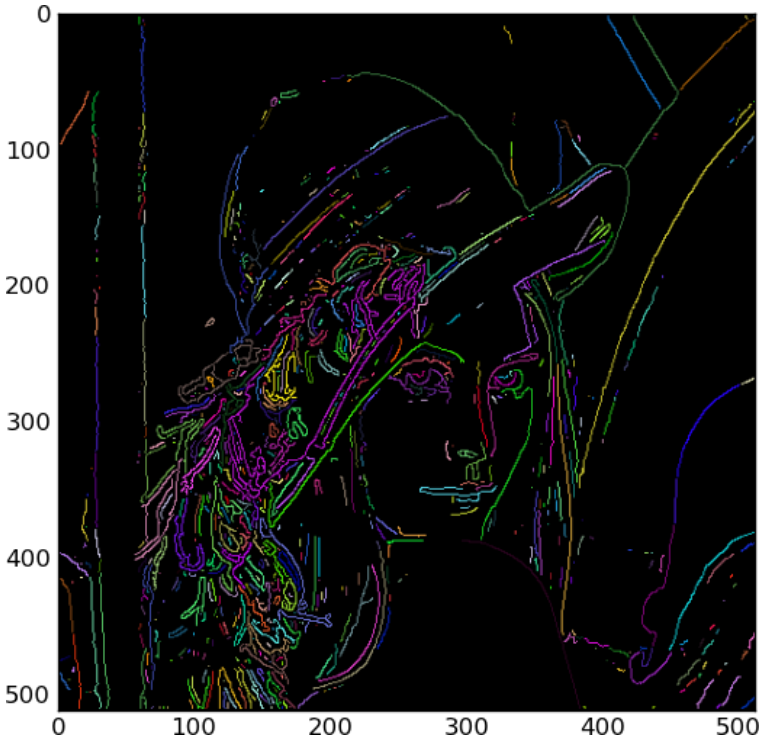
High Threshold (100)

SciPy By Example – Label Connected Components

6. Label connected components using 8-neighborhood (3x3 mask of ones).

Examples/scipy_canny.py

```
39    labels, num_labels = ndi.measurements.label(edge_map, np.ones((3, 3)))
```

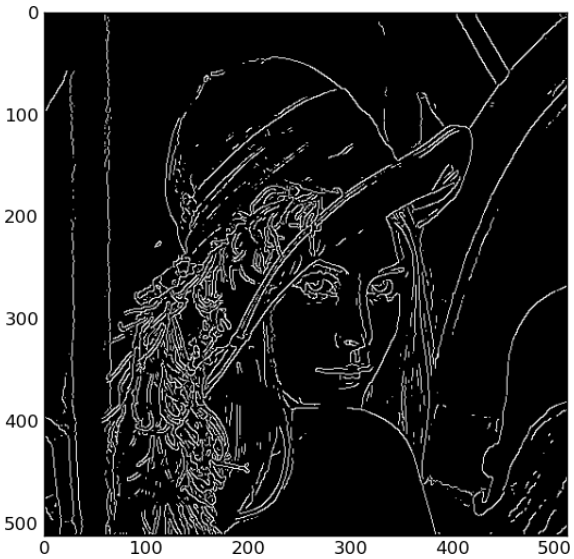


Connected edges with random colors

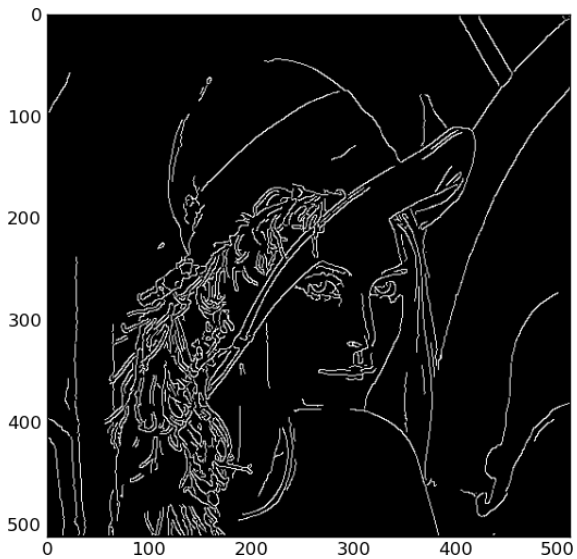
7. Remove connected edges not containing a value above the high threshold.

Examples/scipy_canny.py

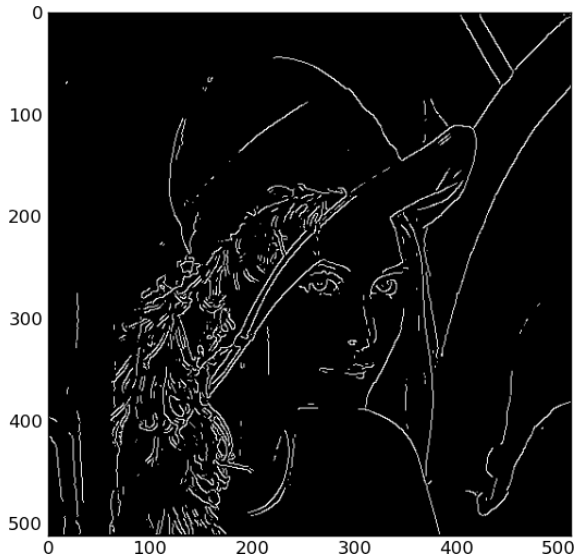
```
40      for i in range(num_labels):
41          if max(mag[labels == i]) < high_thresh:
42              edge_map[labels == i] = False
```



Low Threshold (50)



Hysteresis



High Threshold (100)

SciPy By Example – Non-Max Suppression

- Non-Maximal Suppression of edges produces a binary edge map.
- Gradient magnitude is a maximum in the direction normal to an edge.

Examples/scipy_canny.py

```
 9  def non_maximal_edge_suppresion(mag, orient):
10      """Non Maximal suppression of gradient magnitude and orientation."""
11      # bin orientations into 4 discrete directions
12      abin = ((orient + pi) * 4 / pi + 0.5).astype('int') % 4
13
14      mask = np.zeros(mag.shape, dtype='bool')
15      mask[1:-1, 1:-1] = True
16      edge_map = np.zeros(mag.shape, dtype='bool')
17      offsets = ((1, 0), (1, 1), (0, 1), (-1, 1))
18      for a, (di, dj) in zip(range(4), offsets):
19          cand_idx = np.nonzero(np.logical_and(abin == a, mask))
20          for i, j in zip(*cand_idx):
21              if mag[i, j] > mag[i + di, j + dj] and \
22                 mag[i, j] > mag[i - di, j - dj]:
23                  edge_map[i, j] = True
24      return edge_map
```

Computer Vision in Python

Python Modules for Computer Vision

- This tutorial will focus on directly using Python interfaces to large open source C++ projects:
  - OpenCV
  - ITK
- Several packages for computer vision are built on top of some combination of NumPy, SciPy, OpenCV, and various other modules

SimpleCV	`http://simplecv.org/`
PyVision	`http://sourceforge.net/projects/pyvision/`
PyCVF	`http://pycvf.sourceforge.net/`
PyCam	`http://code.google.com/p/pycam/`
NDVision	`https://launchpad.net/ndvision`

Loading and Saving Images with PIL

- The Python Imaging Library (PIL) has many image manipulation functions.
- PIL is most useful for loading and saving images in a large variety of standard image file formats.
- PIL uses its own image class, but it is easily converted to/from NumPy arrays.

```
import Image
import numpy as np

def load_image(file_name):
    """Read an image file and return a NumPy array."""
    return np.array(Image.open(file_name))

def save_image(data, file_name):
    """Save a NumPy array in an image file."""
    return Image.fromarray(data).save(file_name)
```

What is OpenCV?

- C/C++ library with Python wrappers
- Open source
- Very active community
- Implementations of many computer vision algorithms

OpenCV Functionality Overview

- Graphical User Interface(GUI) - very easy procedural interface
- Input/Output - load/write images and movies in a variety of formats
- Image Processing - filtering,edges,histograms, etc.
- Machine Learning - SVMs, Decision Trees,
- Features
- Object Detection
- and MORE! (http://docs.opencv.org/)

OpenCV By Example – Importing

- Two versions of wrappers: `cv` and `cv2`
- `cv2` contains `cv`
- `cv2` is preferred because it natively uses `numpy`

Examples/opencv_rand.py

```
3  import cv2
4  import numpy as np
```

OpenCV By Example – Display Random Numpy Array

Examples/opencv_rand.py

```
1  #!/usr/bin/python
2
3  import cv2
4  import numpy as np
5
6
7  a = np.random.rand(256, 256)
8  cv2.imshow('Random Image', a)
9  cv2.waitKey()
10 cv2.destroyWindow('Random Image')
```

- Run `Examples/opencv_rand.py`

OpenCV By Example – Display Random Numpy Array

- Generate an array of random numbers with NumPy.

```
7  a = np.random.rand(256, 256)
```

- Show the array as an image in an OpenCV GUI window.
  The string is both the window identifier and the window title.

```
8  cv2.imshow('Random Image', a)
```

- Wait for the user to press a key.
  Optionally this function can take a timeout in milliseconds.

```
9  cv2.waitKey()
```

- Close the window and free memory. This is not strictly required in this case because it happens automatically when the script terminates.

```
10  cv2.destroyWindow('Random Image')
```

Examples/opencv_hello.py

```
1  #!/usr/bin/python
2
3  import cv2
4  import sys
5
6  if len(sys.argv) < 2:
7      # with no arguments, use this default file
8      filename = "/home/tutorial/Tutorial/Data/photo.jpg"
9  elif len(sys.argv) == 2:
10     filename = sys.argv[1]
11 else:
12     sys.exit("Expecting a single image file argument")
13
14 image = cv2.imread(filename)
15 print image.shape
16 image_small = cv2.resize(image, (800, 600))
17 textColor = (0, 0, 255)  # red
18 cv2.putText(image_small, "Hello World!!!", (200, 200),
19             cv2.FONT_HERSHEY_PLAIN, 3.0, textColor,
20             thickness=4, linetype=cv2.CV_AA)
21 cv2.imshow('Hello World GUI', image_small)
22 cv2.waitKey()
23 cv2.destroyAllWindows()
```

OpenCV By Example – Image Processing

- Run `Examples/opencv_hello.py Data/photo.jpg`

OpenCV By Example – Image Processing

- Use OpenCV to read the image file into a NumPy array

```
14  image = cv2.imread(filename)
```

- Resize the image to 800×600 using bilinear interpolation

```
16  image_small = cv2.resize(image, (800, 600))
```

- Define a tuple to represent a color in BGR color space

```
17  textColor = (0, 0, 255)  # red
```

- Draw text on the image in at location (200, 200) at a scale of 3.0

```
18  cv2.putText(image_small, "Hello World!!!", (200, 200),
19              cv2.FONT_HERSHEY_PLAIN, 3.0, textColor,
20              thickness=4, linetype=cv2.CV_AA)
```

OpenCV By Example – Movie Player

Examples/opencv_movie.py

```
1  #!/usr/bin/python
2
3  import cv2
4  import sys
5
6  if len(sys.argv) < 2:
7      # with no arguments, use this default file
8      filename = "/home/tutorial/Tutorial/Data/movie.mov"
9  elif len(sys.argv) == 2:
10     filename = sys.argv[1]
11 else:
12     sys.exit("Expecting a single video file argument")
13
14 movie = cv2.VideoCapture(filename)
15 while True:
16     valid_frame, image = movie.read()
17     if not valid_frame:
18         break
19     cv2.imshow('OpenCV Video Example', image)
20     cv2.waitKey(5)
21
22 cv2.destroyAllWindows()
```

- Run `Examples/opencv_movie.py Data/movie.mov`

OpenCV By Example – Movie Player

- Open the video file

```
14  movie = cv2.VideoCapture(filename)
```

- Start an infinite loop for reading and displaying video frames

```
15  while True:
```

- Read one frame if available, if not exit the loop

```
16      valid_frame, image = movie.read()
17      if not valid_frame:
18          break
```

- Display the image in a GUI window, and wait for 5 milliseconds

```
19      cv2.imshow('OpenCV Video Example', image)
20      cv2.waitKey(5)
```

Examples/opencv_features.py

```
1  #!/usr/bin/python
2
3  import cv2
4  import sys
5
6  if len(sys.argv) < 2:
7      # with no arguments, use this default file
8      filename = "/home/tutorial/Tutorial/Data/photo.jpg"
9  elif len(sys.argv) == 2:
10     filename = sys.argv[1]
11 else:
12     sys.exit("Expecting a single image file argument")
13
14 image = cv2.imread(filename)
15 image_small = cv2.resize(image, (image.shape[1] / 4, image.shape[0] / 4))
16 image_gray = cv2.cvtColor(image_small, cv2.cv.CV_RGB2GRAY)
17 features = cv2.goodFeaturesToTrack(image_gray, 50, .01, 50)
18 features = features.reshape((-1, 2))
19 for x, y in features:
20     cv2.circle(image_small, (x, y), 10, (0, 0, 255))
21 cv2.imshow('OpenCV Features Example', image_small)
22 cv2.waitKey()
23 cv2.destroyAllWindows()
```

OpenCV By Example – Feature Detection

- Run `Examples/opencv_features.py Data/photo.jpg`

OpenCV By Example – Feature Detection

- Convert the color space from RGB to Gray.
  Feature detection requires a gray-scale image.

```
16  image_gray = cv2.cvtColor(image_small, cv2.cv.CV_RGB2GRAY)
```

- Run the "Good Features to Track" algorithm to find 50 points.
  The last two arguments are a threshold and minimum distance between points.

```
17  features = cv2.goodFeaturesToTrack(image_gray, 50, .01, 50)
```

- The features are a 3D array of shape (50, 1, 2).
  Reshape it to (50, 2). The -1 means infer from data length.

```
18  features = features.reshape((-1, 2))
```

- At each feature location (x, y) draw a circle of radius 10 in red.

```
19  for x, y in features:
20      cv2.circle(image_small, (x, y), 10, (0, 0, 255))
```

Exercise 3

OpenCV

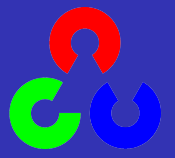
- Open the file `Exercises/ex3.py`.
- Modify the code to run `goodFeaturesToTrack` on the movie.
- Then modify it to run `Canny` edge detection instead.

Exercises/ex3.py

```
16  while True:
17      valid_frame, image = movie.read()
18      if not valid_frame:
19          break
20
21      # Insert code here to process the image before displaying it
22
23      cv2.imshow('Exercise 3: OpenCV', image)
24      cv2.waitKey(5)
```

Exercise 3: Answer

OpenCV



Exercises/Answers/ex3-answer1.py

```
19      image_gray = cv2.cvtColor(image, cv2.cv.CV_RGB2GRAY)
20      features = cv2.goodFeaturesToTrack(image_gray, 50, .01, 50)
21      if features is not None:
22          for x, y in features.reshape((-1, 2)):
23              cv2.circle(image, (x, y), 10, (0, 0, 255))
24      cv2.imshow('Exercise 3: OpenCV', image)
```

Exercises/Answers/ex3-answer2.py

```
19      image_gray = cv2.cvtColor(image, cv2.cv.CV_RGB2GRAY)
20      edges = cv2.Canny(image_gray, 100, 120)
21      cv2.imshow('Exercise 3: OpenCV', edges)
```

Simple ITK

- Introduction to ITK
- Introduction to SimpleITK
- Getting SimpleITK
- Basic Operations
- Advanced Operations
- Building Bridges with Other Libraries

What is ITK ?

- C++ Library
- Open Source (Apache 2.0 License)
- Generic Programming / C++ Templates
- Image Processing
- Image Segmentation
- Image Registration
- ITK's main foci are segmentation and registration, especially in the context of 3D images.

Funding

- Funded (mostly) by the US National Library of Medicine
- With contributions from
  - National Institute of Dental and Craniofacial Research
  - National Science Foundation
  - National Eye Institute
  - National Institute of Neurological Disorders and Stroke
  - National Institute of Mental Health
  - National Institute on Deafness and Other Communication Disorders
  - National Cancer Institute

History

- Started in 2000
- Developed by Companies and Universities
  - GE Corporate Research
  - Insightful
  - Kitware
  - UNC Chapel Hill
  - University of Pennsylvania
  - University of Utah

History

- Recent contributions by a larger community
  - Harvard - Brigham and Women's Hospital
  - University of Iowa
  - Georgetown University
  - INRA - France
  - German Cancer Research Center
  - . . . and many others . . .

Project Profile

- Total lines of code: 1,886,674
- Active developers: 56 (past 12 months)
- Total developers: 146 (full history of the project)
- Top 2% largest open source teams in the world
- Estimated cost of development: \$34.5 M[1]

[1] Ohloh: `http://www.ohloh.net/p/itk`

Funding

- First 10 years of development = $15M
- Refactoring of ITKv4 in 2011 = $5M
- Yearly maintenance supported by NLM

Simple ITK

SimpleITK provides a template-free layer on top of ITK that is automatically wrapped for Python.

More info: `http://simpleitk.org`

Installing Simple ITK

- Two main options are building from source with Python wrappings
- Or easy_install (easy_install is much easier).

```
bash ~/ easy_install SimpleITK  # It's that easy (you may have to be root).
```

- Let's walk through an example of basic Simple ITK usage from `Examples/sitk_intro.py`.

Creating Images with Simple ITK

- It's very easy to create images using SimpleITK.
- The images will be two-dimensional or three-dimensional based on the constructor

Examples/sitk_intro.py

```
4  import SimpleITK as sitk
5
6  # Create Images
7  image = sitk.Image(256, 256, 256, sitk.sitkInt16)
8  twoD = sitk.Image(64, 64, sitk.sitkFloat32)
```

Creating Images with Simple ITK

- sitk is the module
- Image is the constructor for the Image class
- Height, width, depth (omit depth for 2D images)
- Datatype (more on this later)

What just happened?

Examples/sitk_intro.py

```
11  print image.GetPixel(0, 0, 0)
12  image.SetPixel(0, 0, 0, 1)
13  print image.GetPixel(0, 0, 0)
```

- Get the voxel value at [0,0,0]?
- Hmm, I don't like it, so set to 1
- What is the value at [0,0,0] now?

What just happened?

Examples/sitk_intro.py

```
16  print image[0, 0, 0]
17  image[0, 0, 0] = 10
18  print image[0, 0, 0]
```

Without warning, we sprinkled syntactic sugar on you!

- `image[0,0,0]` is shorthand for `image.GetPixel(0,0,0)`
- `image[0,0,0] = 10` is shorthand for `image.SetPixel(0,0,0,10)`

SimpleITK Pixel Types

- Simple ITK supports a wide array of pixel types
- To get a human readable assessment of the pixel type:

Examples/sitk_intro.py

```
21 print image.GetPixelIDTypeAsString()
```

SimpleITK Image Summary

- Images are created using `SimpleITK.Image ( w, h, d, Type )`
- Images can be 2- or 3-dimensional
- Images can describe themselves
- Images have simple pixel accessors

Filtering with SimpleITK

- Basic operations are very simple
- Let's look at an image with a Gaussian subtracted from it:

Examples/sitk_intro.py

```
22 smooth = sitk.SmoothingRecursiveGaussian(image, 2.0)
```

- The output of SmoothingRecursiveGaussian is of type float
- The input image is signed short
- Most SimpleITK filters with 2 inputs require the same type
- We need to cast in order to difference the input image with the smoothed image

Introducing Cast

- The Cast Filter works as you would expect.

Examples/sitk_intro.py

```
25 print "Before: ", smooth.GetPixelIDTypeAsString()
26 smooth = sitk.Cast(smooth, image.GetPixelIDValue())
27 print "After: ", smooth.GetPixelIDTypeAsString()
28 sitk.Show(sitk.Subtract(image, smooth), "DiffWithGaussian")
```

Sizes and Indices

- Extracting a region of interest from a SimpleITK image is easy with `sitk.Extract`.

Examples/sitk_intro.py

```
31  size = [64, 64, 0]
32  start = [64, 0, 0]
33  sitk.Show(sitk.Extract(image, size, start), "Extracted")
```

More Advanced Filtering

- Now that we can do basic filtering, let's look at more advanced filters
- How about Otsu Thresholding?

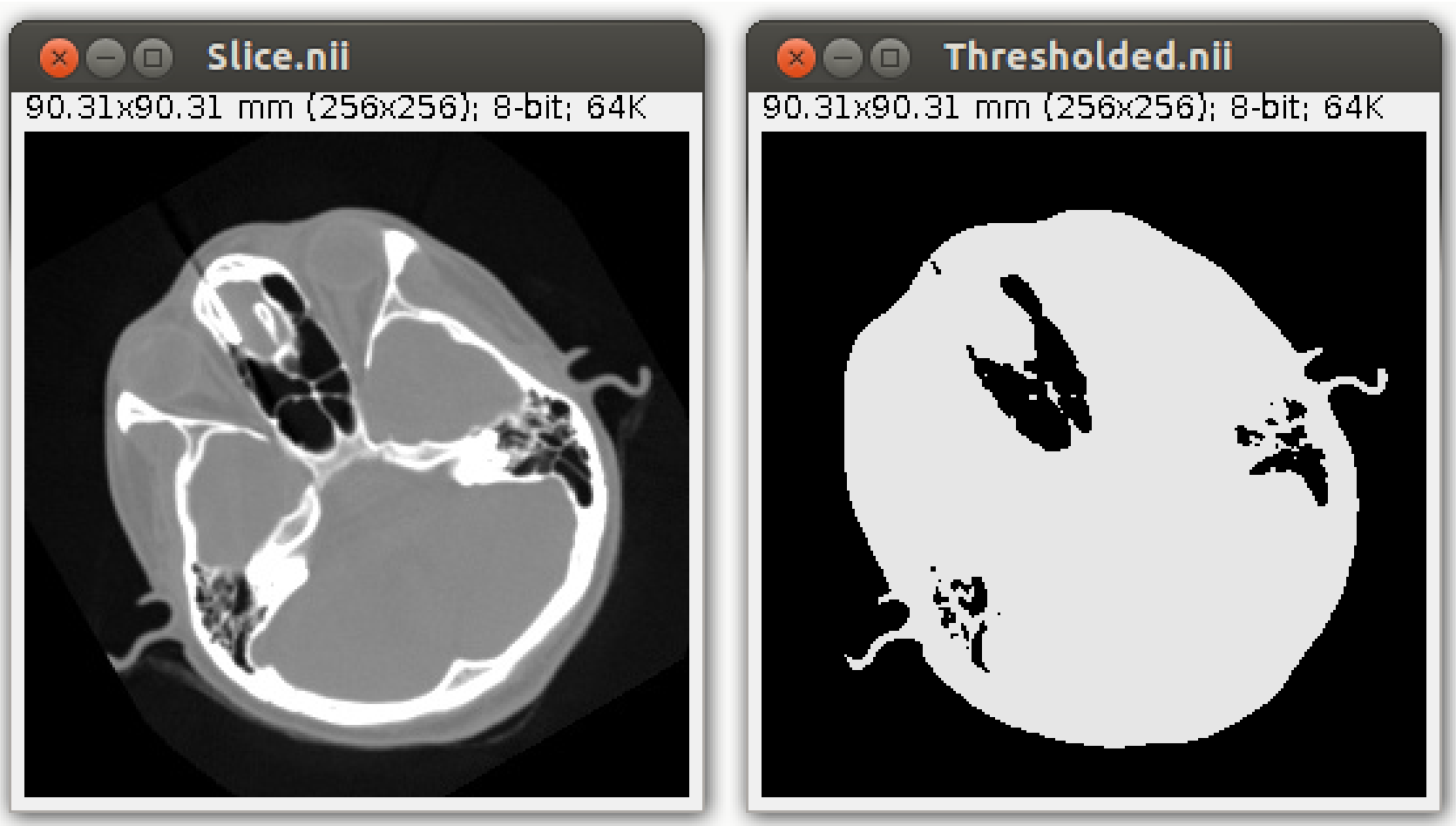
Examples/sitk_example.py

```
11  import SimpleITK as sitk
12
13  image = sitk.ReadImage(filename)
14  sitk.Show(image, 'Slice')
15  image = sitk.Cast(image, sitk.sitkFloat32)
16  thresh = sitk.OtsuThreshold(image, 0, 230, 15)
17  sitk.Show(thresh, 'Thresholded')
```

- Notice that filters can also be instantiated as objects

More Advanced Filtering

- We can run nearly every filter from ITK in this manner and quickly view results (as shown below).
- Bonus: `sitk.Show` allows us to view 3D Images with ImageJ



Integration with NumPy

- Let's look at integrating SimpleITK with NumPy.

Examples/sitk_numpy.py

```
11  import SimpleITK as sitk
12  import numpy as np
13  import pylab
14
15  image = sitk.ReadImage(filename)
16
17  a = sitk.GetArrayFromImage(image)
18
19  h, bins = np.histogram(a, bins=255, range=(0, 255))
20
21  mode = 0
22  for i in range(1, h.shape[0]):
23      if h[i] > h[mode]:
24          mode = i
25  print "Mode of image: " + str(mode)
26
27  # Print histogram
28  pylab.figure()
29  pylab.hist(a.flatten())
30  pylab.show()
```

Exercise 4

SimpleITK

- Open the file `Exercises/ex4.py`.
- Let's take the previous examples and build on them.
  1. Use `sitk.RecursiveGaussian` to smooth the image, then plot the histogram with Matplotlib.
  2. Use `sitk.Threshold` to segment bone from the input image.
- Hint: Bone has intensity greater than 210 in this image.

Exercise 4: Answer

SimpleITK

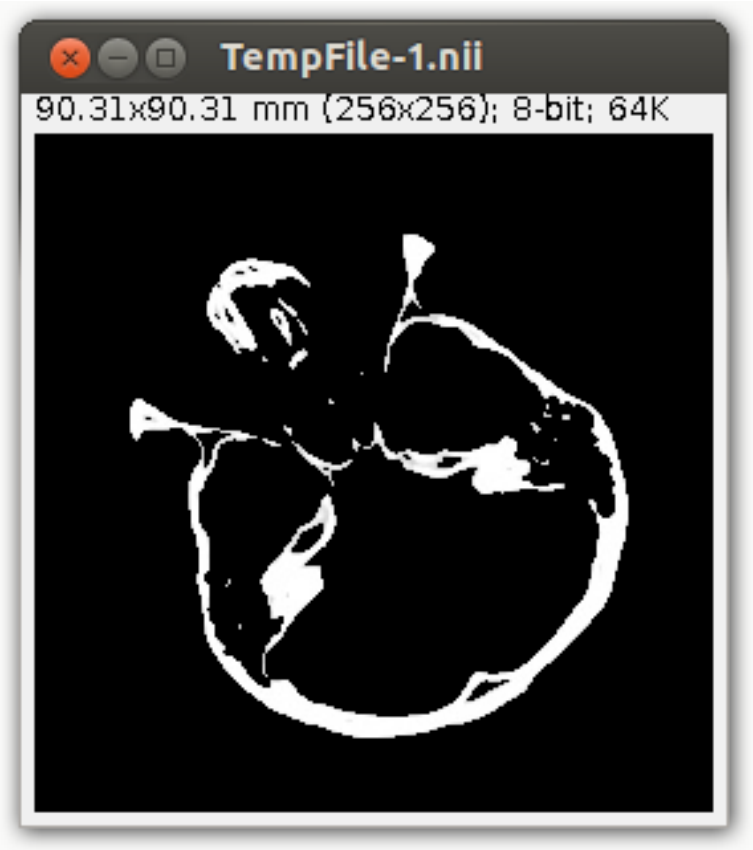
- Great Job! your code should look something like this

Exercises/Answers/ex4-answer.py

```
11  import SimpleITK as sitk
12  import pylab
13
14  image = sitk.ReadImage(filename)
15  image_array = sitk.GetArrayFromImage(image)
16
17  sitk.Show(image)
18
19  blurred = sitk.RecursiveGaussian(image, 4)
20  blurred_array = sitk.GetArrayFromImage(blurred)
21
22  # Generate a normal and blurred histogram
23  pylab.figure()
24  pylab.hist(image_array.flatten())
25  pylab.figure()
26  pylab.hist(blurred_array.flatten())
27  pylab.show()
28
29  # Threshold the image properly
30  thresh = sitk.Threshold(image, 210, 255)
31
32  sitk.Show(thresh)
```

More Advanced Filtering

- Your bone segmentation should look something like this.



Credits

Material taken from the following tutorials:

- The SimpleITK tutorial from MICCAI 2011 (`https://github.com/SimpleITK/SimpleITK-MICCAI-2011-Tutorial`).
- The ITKv4 next generation tutorial (`https://github.com/InsightSoftwareConsortium/ITKv4-TheNextGeneration-Tutorial`).

Parallel Processing in Python

Multi-Threading in Python

- Python supports multi-threading of code but has an important limitation.
- Only one thread can execute Python code at a time.
- Multi-thread can still be useful in certain cases such as
  - running multiple I/O-bound tasks simultaneously
  - code that uses a lot of C-based modules (e.g. NumPy and SciPy)
- To get parallel processing in Python it is often better to use multi-processing.

http://docs.python.org/library/threading.html

Multi-Processing in Python

- The `multiprocessing` module provides convenient ways to divide Python execution over multiple processes.
- Create a pool of worker processes and divide up the work.

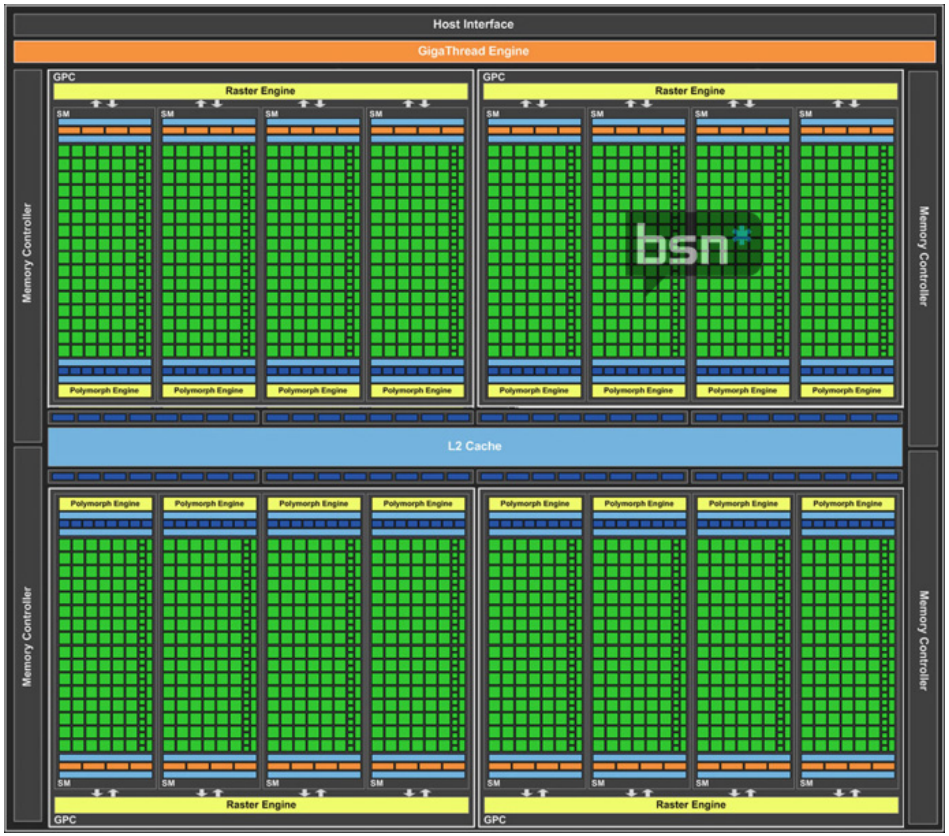
Examples/multiproc.py

```
1  from multiprocessing import Pool
2
3  def f(x):
4      return x*x
5
6  if __name__ == '__main__':
7      pool = Pool(processes=4)              # start 4 worker processes
8      result = pool.apply_async(f, [10])    # evaluate "f(10)" asynchronously
9      print result.get(timeout=1)           # prints "100" unless your computer is slow
10     print pool.map(f, range(10))          # prints "[0, 1, 4,..., 81]"
```

http://docs.python.org/library/multiprocessing.html

Modern GPGPU

- GPUs have been evolving from dedicating graphics rendering hardware, to powerful general computing devices:



Modern GPU Architecture (Nvidia Kepler)

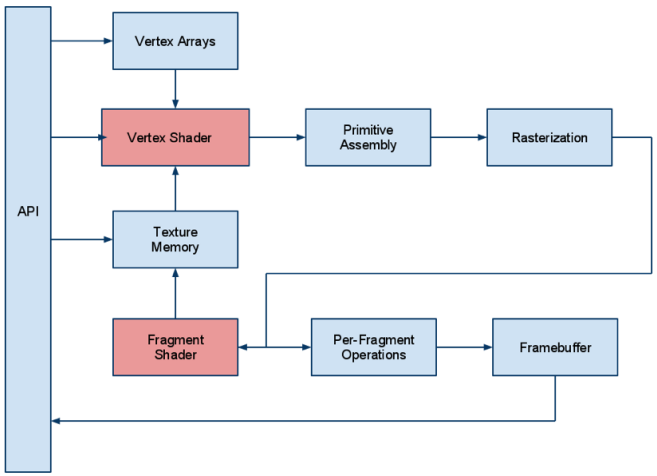
http://blogs.nvidia.com/tag/kepler/index.html

Modern GPGPU

- Massive amount of processing cores (3072 CUDA cores in Nvidia GeForce GTX 690 single card)
- Each core has its own dedicated register file
- Each cluster of cores share L1 cache
- Clusters of cores share L2 cache
- Huge memory throughput (512bit, 384GB/sec for Nvidia GeForce 690)
- Atomic operations
- 64-bit precision
- Error-correcting Code (ECC) memory to meet HPC requirement
- GPU Vs CPU on HPC performance
  - Most CV developers: 10 to 500x speed-up depending on the application and implementation
  - Intel: "GPUs are ONLY up to 14 times faster than CPU."

http://blogs.nvidia.com/tag/kepler/index.html

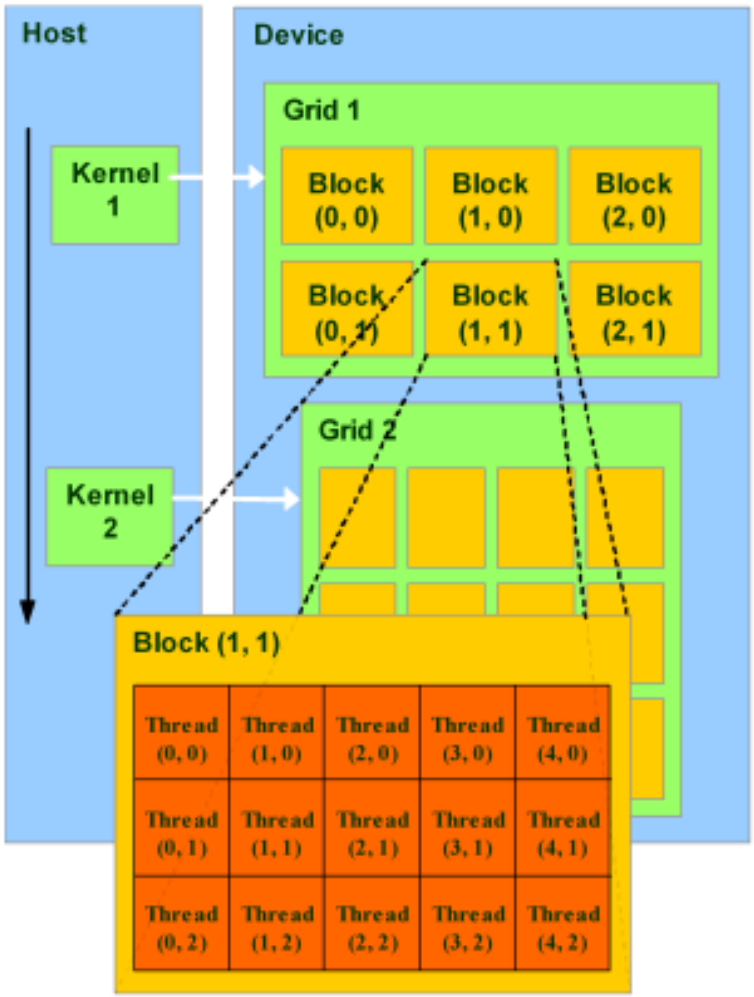
GPGPU Programming Model: Shading Language



Shading Language (Vertex & Fragment Shader)

- Map parallel computing task to openGL rendering pipeline
- Pack input data as vertices and texture, operate these data from vertex shader and fragment shader, results are output to frame buffer
- Pros:
  - enable the GPU for general use for the first time
- Cons:
  - single precision
  - no flow control and branching
  - very limited kernel size
  - lack of cross-kernel synchronization

GPGPU Programming Model: CUDA and OpenCL



CUDA Programming Model
(OpenCL has very similar model)

- Enable C/C++ like direct parallel thread programming
- Enable cross-thread synchronization
- Enable direct memory access
- Enable atomic operations
- Effectively remove the kernel size limit
- Enable full support for flow control and branching
- GPUs become truly general purpose

http://ixbtlabs.com/articles3/video/cuda-1-p5.html

GPGPU in Computer Vision

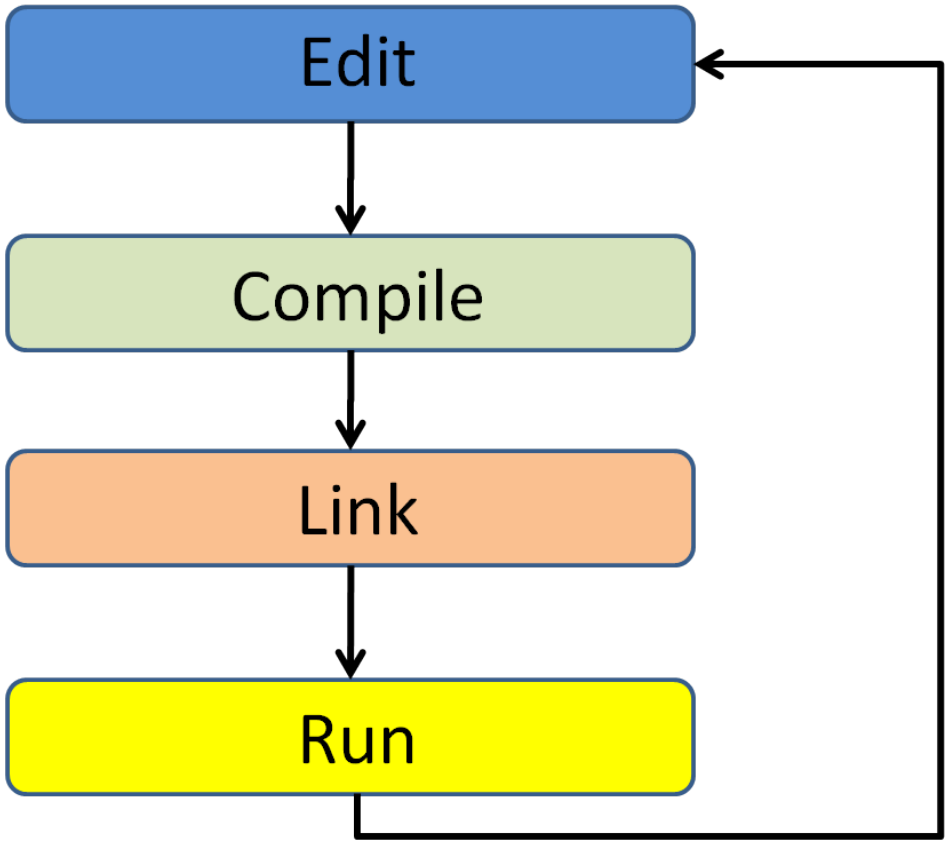
- Many hardwired graphics rendering resources in GPU, including texture engine, 3D transformation, make GPU a natural fit for computer vision and computational photography processing.
- OpenVIDIA: Parallel GPU Computing Vision Library developed by Nvidia
- OpenCV: more and more GPU Modules
- Nvidia CUDA Zone: An Image and Computer Vision GPU library, including segmentation, feature processing, stereo imaging, machine learning and data processing

Why use GPGPU in Python?

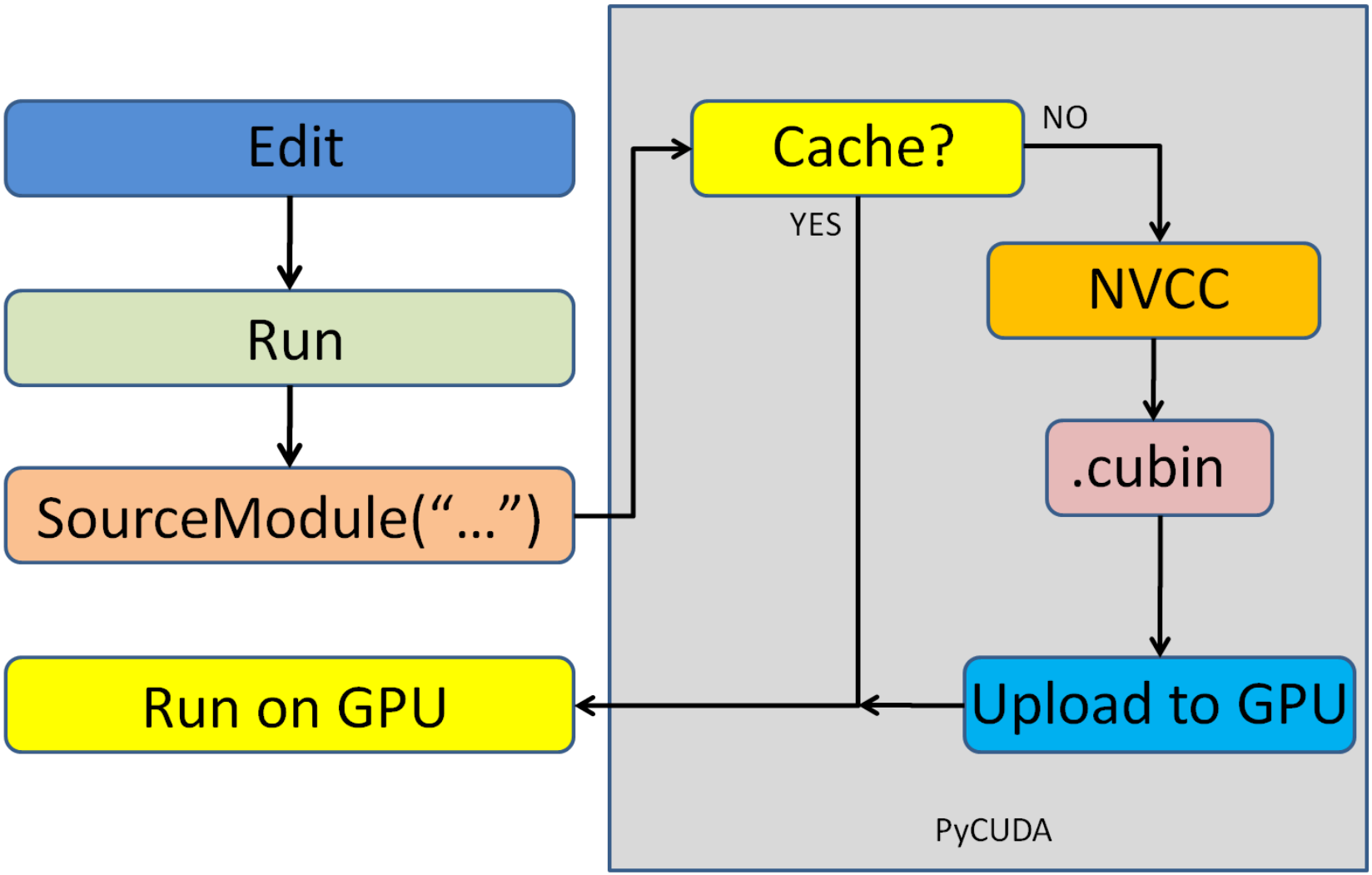
- GPUs are everything that scripting languages are not
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum performance
- GPU and Python can be complement to each other
  - Use Python script as the brain, organizing processing module and data connection
  - Use GPU as muscles, executing intensive, dirty inner loops
- Play to the strengths of each programming environment
- Python + CUDA: PyCUDA
- Python + OpenCL: PyOpenCL

http://mathema.tician.de/software/pycuda

Native CUDA work flow



PyCUDA work flow



http://mathema.tician.de/software/pycuda

PyCUDA Example

```
import pycuda.driver as cuda                          # import PyCUDA module
import pycuda.autoinit, pycuda.compiler               # initialize the CUDA hardware
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)# create a 4x4 array random numbers
a_gpu = cuda.mem_alloc(a.nbytes)                      # alloc the same sized array in GPU
cuda.memcpy_htod(a_gpu, a)                            # copy the data from host to device

mod = pycuda.compiler.SourceModule("""                # specify a CUDA kernel function
    __global__  void  twice (float *a)
    {
        int idx = threadIdx.x + threadIdx.y*4;
        a[idx] *= 2;
    }
    """)
func = mod.get_function("twice")                      # get the handle of CUDA kernel
func(a_gpu, block=(4,4,1))            # run the CUDA function with a configuration

a_doubled = numpy.empty_like(a)                       # create an empty buffer on host
cuda.memcpy_dtoh(a_doubled, a_gpu)                    # copy result from device to host
print a_double
print a
```

http://mathema.tician.de/software/pycuda

gpuarray: Simple Linear Algebra

- Meant to look and feel just like numpy
- `+`, `-`, `*`, `/`, `fill`, `sin`, `exp`, `rand`, basic indexing, `norm`, inner product, ...
- Mixed types (`int32 + float32 = float64`)
- "`print gpuarray`" for debugging
- Allows access to raw bits

`http://mathema.tician.de/software/pycuda`

gpuarray: Element-wise expressions

- An example of element-wise operation in PyCUDA:

```
from pycuda.curandom import rand as curand     # import CUDA random number module

a_gpu = curand((50,))                          # create a 1-d array with random number
b_gpu = curand((50,))

from pycuda.elementwise import ElementwiseKernel  # import ElementwiseKernel module

# specify the detail of element-wise operation
lin_comb = ElementwiseKernel(
    " float a, float *x, float b, float *y, float *z",
    "z[i] = a*x[i] + b*y[i]")

c_gpu = gpuarray.empty_like(a_gpu)             # create a GPU array of same size
lin_comb(5, a_gpu, 6, b_gpu, c_gpu)            # run the ElementwiseKernel function
assert  la.norm((c_gpu -  (5*a_gpu+6*b_gpu)).get()) < 1e-5

print a_gpu
print b_gpu
print c_gpu
```

http://mathema.tician.de/software/pycuda

gpuarray: Reduction made easy

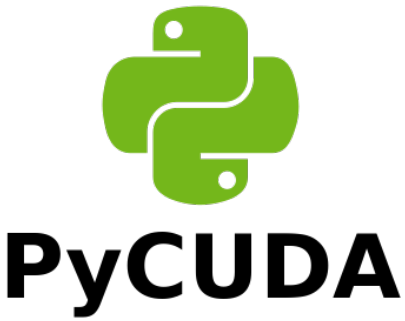
- Example: A scalar product calculation

```
from pycuda.reduction import ReductionKernel  # import ReductionKernel module

# specify the detail of the reduction operation
dot = ReductionKernel(
        dtype_out=numpy.float32,
        neutral="0",
        reduce_expr="a+b",
        map_expr="x[i]*y[i]",
        arguments="const float *x, const float *y")

from pycuda.curandom import rand as curand

x = curand((1000*1000), dtype=numpy.float32)
y = curand((1000*1000), dtype=numpy.float32)

x_dot_y = dot(x, y).get()
x_dot_y_cpu = numpy.dot(x.get(), y.get())

print x
print y
print x_dot_y
print x_dot_y_cpu
```

http://mathema.tician.de/software/pycuda

PyCUDA: Vital Information

- http://mathema.tician.de/software/pycuda
- Complete documentation
- MIT License (no warranty, free for all use)
- Requires: numpy, Python 2.4+ (Win/OS X/Linux)
- Support via mailing list



http://mathema.tician.de/software/pycuda

Introducing... OpenCL and PyOpenCL

- OpenCL: the Open Computing Language
  - Vendor-independence
  - Single abstraction works well for GPUs, CPUs
  - A JIT C compiler baked into a library
  - Intel's future integrated GPUs will be supporting (more ubiquitous, available in cloud)
- PyOpenCL: the "PyCUDA" for OpenCL
  - Complete, mature API wrapper
  - Has: Arrays, elemet-wise operations, ...
  - http://mathema.tician.de/software/pyopencl

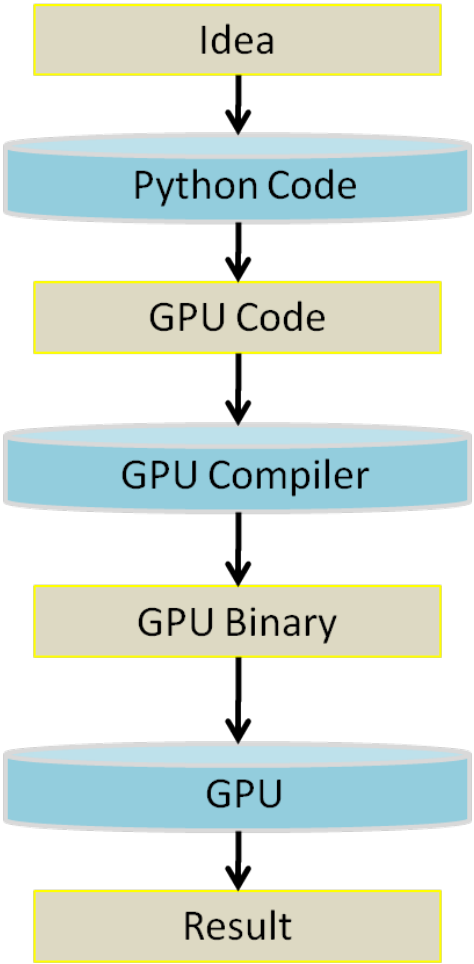
http://mathema.tician.de/software/pyopencl

OpenCL: Same flavor, different recipe

```
import pyopencl as cl , numpy      # import PyOpenCL module

a = numpy.random.rand(50000).astype(numpy.float32)  # create a buffer on host

ctx = cl.create_some_context()     # create an OpenCL context
queue = cl.CommandQueue(ctx)

# create a buffer on device and copy the data in host buffer to device buffer
a_buf = cl.Buffer(ctx , cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_buf , a)

# specify an OpenCL kernel function
prg = cl.Program(ctx, """
      __kernel void twice(__global float *a)
      {
          int gid = get_global_id (0);
          a[gid] *= 2;
      }
      """).build()

# run an OpenCL kernel function
prg.twice(queue, a.shape, None, a_buf).wait()
```

http://mathema.tician.de/software/pyopencl

Metaprogramming



- GPU code does not need to be a compile-time constant
- Code is data - it wants to be reasoned about at run time
- Automated tuning
- Data types
- Specialized code for given problem
- Constants faster than variables
- Loop unrolling

http://mathema.tician.de/software/pycuda

Showcase...

Interfacing Python with Other Languages

Wrapping C/C++ code in Python

There are several options for wrapping C/C++ code in Python

- Calling the Python C API Manually (not for the faint of heart)
- Interfacing with shared libraries using ctypes
- Using Cython to generate C API calls (good for optimizing critical pieces of code)
- Using SWIG to generate C API calls (good for wrapping larger libraries)
- Using Boost.Python to bridge between Python and C++

Working with the Python C API

- Calling the C API directly allows you to get the most performance possible and have the greatest control of execution.
- This is also one of the least easily maintainable methods.
- For more information, see `http://docs.python.org/extending/extending.html`.
- There is a simple example at `Examples/c-api-demonstration/cdemo.c`

A Note on the ctypes Module

- ctypes is a built-in module that allows calling functions from shared libraries.
- A more exhaustive explanation of what ctypes can do can be found here: http://docs.python.org/library/ctypes.html.
- For example, we can easily call the standard libc time function with the following snippet of code:

```
7  from ctypes import cdll, CDLL
8
9  # The name of our library
10 library_name = 'libc.so.6'
11
12 # Load the library as a module
13 cdll.LoadLibrary(library_name)
14 libc = CDLL(library_name)
15
16 # Call the standard time function and print the results
17 print libc.time(None)
```

- This can be very useful, but doesn't feel as "pythonic" as some of the other approaches.

Cython

- Cython is a tool for "compiling" Python-like code into C to be compiled into python extension modules.
- There are sets of annotations that can further optimize code through static typing, removal of bounds checking, etc.
- An example of a `pyx` file and generated C file can be found in `Examples/cython`
- Let's go through a simple example.

More Info: `http://www.cython.org/`

Cython

- This simple code compiled with Cython is 35% faster.

```
0  def f(x):
1      return x**2-x
2
3  def integrate_f(a, b, N):
4      s = 0
5      dx = (b-a)/N
6      for i in range(N):
7          s += f(a+i*dx)
8      return s * dx
```

- By adding type declarations we can go a great deal faster, even with this simple example.

Cython

- This code compiled with Cython is 400% faster, but we lose compatibility with Python through the calls to `cdef`.

```
0   def f(double x):
1       return x**2-x
2
3   def integrate_f(double a, double b, int N):
4       cdef int i
5       cdef double s, dx
6       s = 0
7       dx = (b-a)/N
8       for i in range(N):
9           s += f(a+i*dx)
10      return s * dx
```

- We can do even better by typing the function call.

Cython

- This code compiled with Cython is 15,000% faster, but the calls are no able to raise Python errors.

```
0   cdef double f(double x) except? -2:
1       return x**2-x
2
3   def integrate_f(double a, double b, int N):
4       cdef int i
5       cdef double s, dx
6       s = 0
7       dx = (b-a)/N
8       for i in range(N):
9           s += f(a+i*dx)
10      return s * dx
```

Cython

- Let's take a look at that timing on our virtual machines using

```
In [1]: import simple

In [2]: import simplec

In [3]: import simple_opt

In [4]: import simple_opt_full

In [5]: %timeit for x in range(1,1001): simple.integrate_f(3, 5, x)
10 loops, best of 3: 137 ms per loop

In [6]: %timeit for x in range(1,1001): simplec.integrate_f(3, 5, x)
10 loops, best of 3: 79.8 ms per loop

In [7]: %timeit for x in range(1,1001): simple_opt.integrate_f(3, 5, x)
10 loops, best of 3: 55.6 ms per loop

In [8]: %timeit for x in range(1,1001): simple_opt_full.integrate_f(3, 5, x)
1000 loops, best of 3: 1.85 ms per loop
```

Cython

- Here is an example of how to compile your cython into a loadable module (taken from `http://docs.cython.org/src/reference/compilation.html`).

```
0  $ cython yourmod.pyx
1  $ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing \
2  -I/usr/include/python2.7 -o yourmod.so yourmod.c
```

- This should give you an idea of the possibilities for rapid optimization with Cython. See `http://cython.org/` for more information.

SWIG

- SWIG is a great option for wrapping existing C/C++ code for many different languages, including Python
- An example of swig usage can be found in `Examples/swig`
- SWIG is generally used for large projects that need semi-automated wrapping for multiple languages (SimpleITK uses this for its wrapping).

More Info: `http://www.swig.org/`

SWIG

- Here is an example of a C file we want to automatically wrap with SWIG

```
3  #include <time.h>
4  double My_variable = 3.0;
5
6  int fact(int n) {
7    if (n <= 1) return 1;
8    else return n*fact(n-1);
9  }
10
11 int my_mod(int x, int y) {
12   return (x%y);
13 }
14
15 char *get_time()
16 {
17   time_t ltime;
18   time(&ltime);
19   return ctime(&ltime);
20 }
```

SWIG

- In SWIG we define the interface or `.i` files

```
2   %module example
3   %{
4   /* Put header files here or function declarations like below */
5   extern double My_variable;
6   extern int fact(int n);
7   extern int my_mod(int x, int y);
8   extern char *get_time();
9   %}
10
11  extern double My_variable;
12  extern int fact(int n);
13  extern int my_mod(int x, int y);
14  extern char *get_time();
```

SWIG

- To generate our example wrapping and build example.c itself, we run the following commands.

```
swig -python example.i
gcc -c example.c example_wrap.c -I/usr/include/python2.7
ld -shared example.o example_wrap.o -o _example.so
```

- Swig generates a shared library and python wrapper for importing it as shown below.

```
In [9]: import example

In [10]: example.fact(10)
Out[10]: 3628800
```

SWIG

- This pattern can be used to wrap existing C and C++ libraries in a maintainable way.
- There is also support for converting between STL types and Python types.

Boost.Python

- Boost.Python is an alternative to SWIG.
  - Only for making Python bindings (not multi-language)
  - Designed to make wrapping of C++ easy.
  - Handles C++ classes, virtual functions, inheritance, etc.
  - Only involves writing C++ code (no interface files)
- To wrap the previous example code

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(example)
{
    using namespace boost::python;
    scope().attr("My_variable") = My_variable;
    def("fact", fact);
    def("my_mod", my_mod);
    def("get_time", get_time);
}
```

More Info: http://www.boost.org/doc/libs/1_49_0/libs/python/doc/

Interfacing with Matlab

Interfacing with Matlab

Any transition to Python from Matlab will require a means to utilize your previous Matlab code base. There are several methods for doing this:

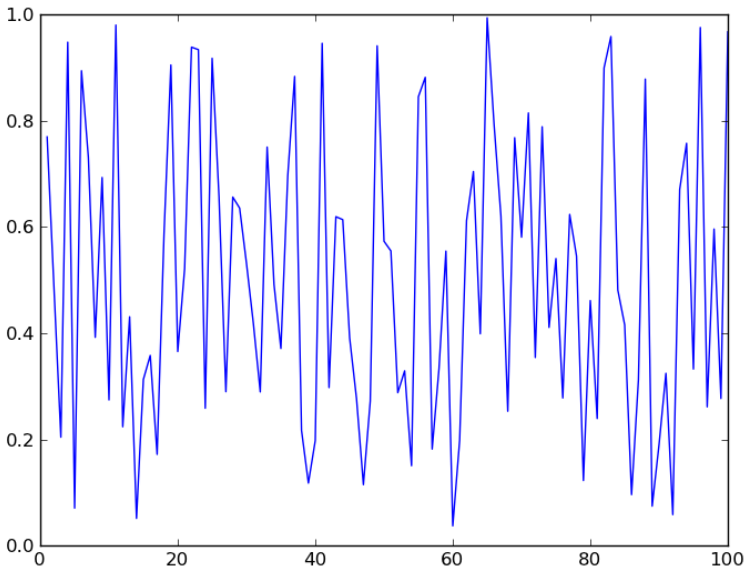
1. Converting Matlab Code to Python
   - Open-Source Matlab-to-Python Compiler (OMPC) (`ompc.juricap.com`):
     - Converts existing Matlab functions to Python, does not require Matlab license, but only works for simple functions
2. File Interchange with Matlab
   - Load/Save .mats in Python:
     - Scipy.io.< *loadmat*/*savemat* > load/save .mats, does not require Matlab license, but may need to manipulate format after loading
3. Calling Legacy MATLAB Code from Python
   - Mlabwrap(`mlabwrap.sourceforge.net`):
     - Calls Matlab functions in manner similar to Matlab, does require Matlab license, and slower than Matlab

OMPC Example

Convert a Simple Matlab function that plots random numbers to Python

Convert .m to .pym

```
In [11]: import ompc
In [12]: import plot_rand_nums_ex1
In [13]: plot_rand_nums_ex1(100)
```



Examples/plot_rand_nums_ex1.m

```
1  function plot_rand_nums_ex1(maxX)
2  %Function to plot a vector of random numbers
3  x= 1:maxX;
4  randx= rand(1,maxX);
5
6  plot(x,randx)
```

Examples/plot_rand_nums_ex1.pym

```
1  # This file was automatically translated
2  # by OMPC (http://ompc.juricap.com)
3
4  from ompc import *
5
6  @mfunction("")
7  def plot_rand_nums_ex1(maxX=None):
8      #Function to plot a vector of random numbers
9      x = mslice[1:maxX]
10     randx = rand(1, maxX)
11
12     plot(x, randx)
```

http://ompc.juricap.com/download

Load/Save .mat Examples

Create Structure in Matlab, save as .mat, and load into Python

MATLAB

```
>> xx.range= 1:11;
>> xx.name= 'vars1';
>> xx.cellvalues= {[15 16],'var1',[17 18],'var2'}
xx =
          range: [1x11 double]
           name: 'vars1'
     cellvalues: {1x4 cell}
>> save('xx.mat','xx')
```

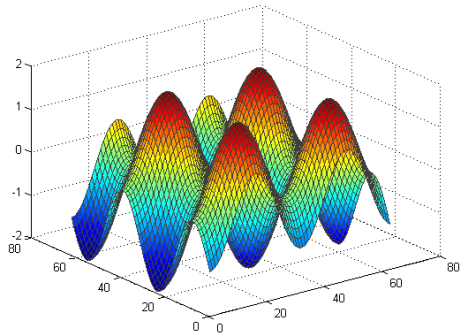
Python

```
> import scipy.io as sio
> xx= sio.loadmat('xx.mat')
> print xx
{'xx': array([[ ([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]], [u'vars1'],
[[array([[15, 16]], dtype=uint8), array([u'var1'],
      dtype='<U4'), array([[17, 18]], dtype=uint8), array([u'var2'],
      dtype='<U4')]])]],
      dtype=[('range', '|O4'), ('name', '|O4'), ('cellvalues', '|O4')]),
      '__version__': '1.0', '__header__': 'MATLAB 5.0 MAT-file, Platform: PCWIN64,
      Created on: Tue May 29 12:53:43 2012', '__globals__': []}
```

Mlabwrap Example

- Call Matlab's Surf() function from Python

```
1  #!/usr/bin/python
2  # Copyright 2011, Kitware, Inc.
3  # http://mlabwrap.sourceforge.net/
4  ''' Examples for calling matlab functions from Python'''
5
6  from mlabwrap import mlab; from numpy import *
7
8  def main():
9      '''Calling Matlab's surf() command'''
10     xx = arange(-2*pi, 2*pi, 0.2) #Define range of inputs
11     mlab.surf(subtract.outer(sin(xx),cos(xx))) #call matlab's surf() function
```



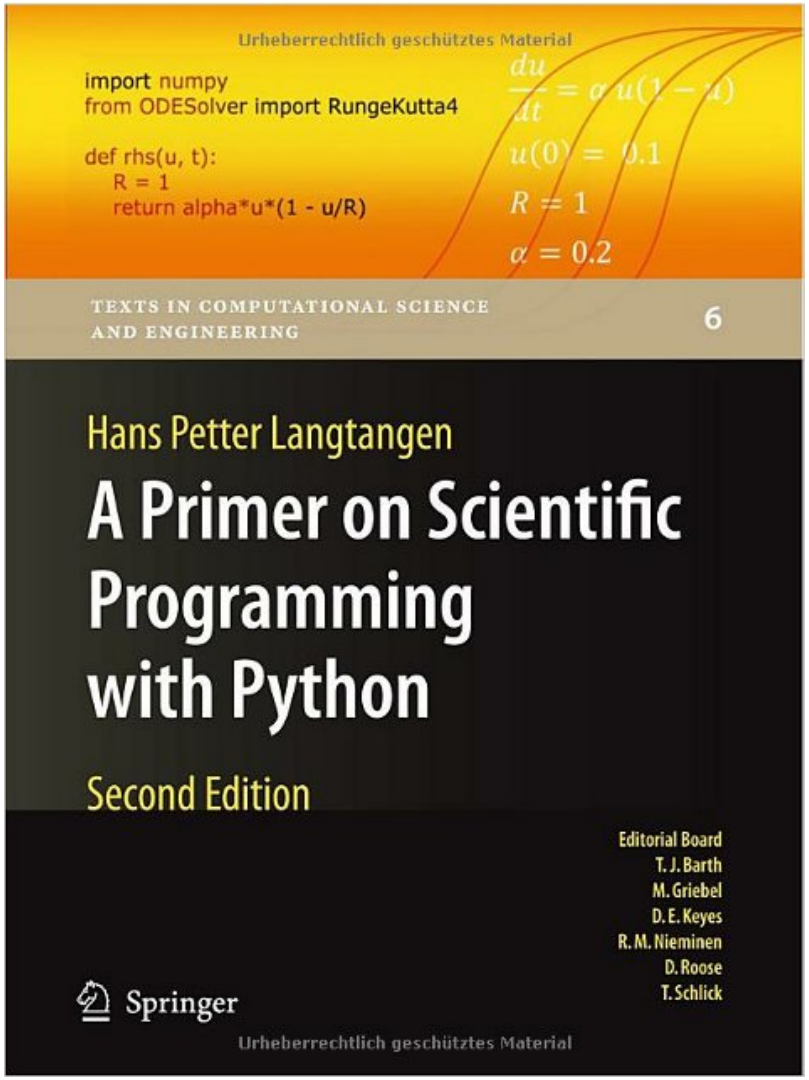
- Call Matlab's Singular Value Decomposition (svd) function

```
13     '''Calling Matlab's Singular Value Decomposition svd() command'''
14     U, S, V = mlab.svd([[1,2],[1,3]], nout=3)
```

More Info: http://mlabwrap.sourceforge.net/

Wrap-Up

Raffle Prize 1



Book: A Primer on Scientific Programming with Python

Raffle Prize 2

ASUS Transformer Pad TF300

Send Us Feedback

Please send us feedback, either positive or negative.

Thank You!






matt.leotta amitha.perera patrick.reynolds eran.swears yongzhao varun

@kitware.com @google.com

Appendix

Native Installation of Scientific Python

Native Installation of Scientific Python

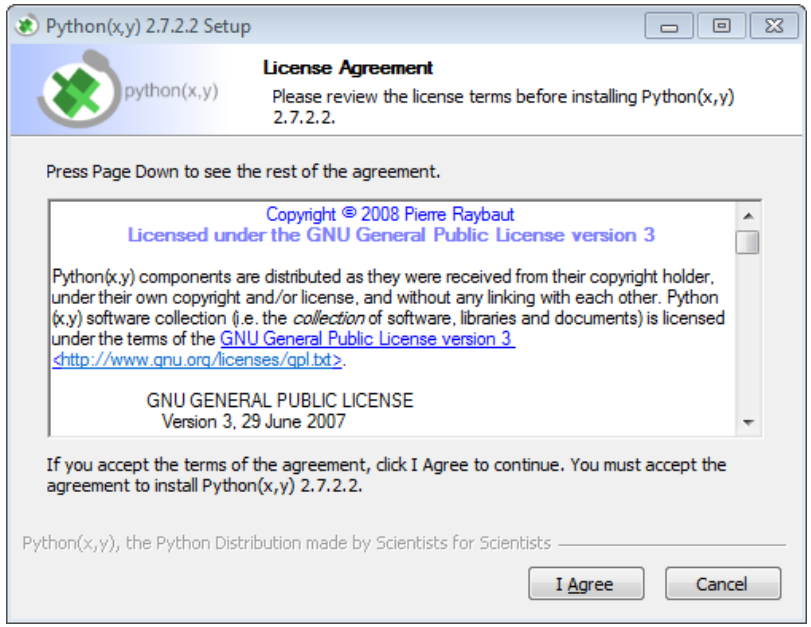
- The virtual machine provides a nice way to try out software without worrying about installation.
- Eventually you'll want to install these toolkit in your native OS.
- The following slides provide basic installation instructions for
  - Windows
  - Mac OS X
  - Linux

Installing on Windows

- It is difficult to install individual toolkits on Windows.
- Packages installed independently often do not work together because
  - package versions may be incompatible with each other
  - packages are compiled with different compilers.
- Lucky there is a project called Python(x,y) which installs almost everything you need for scientific Python and configures it to work together.

Installing on Windows – Python(x,y)

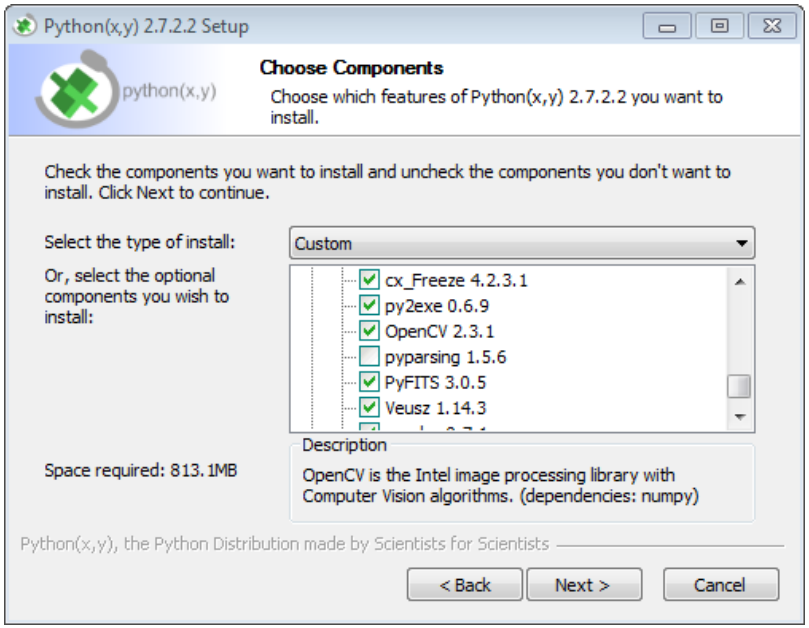
1. Download the package installer (currently Python(x,y)-2.7.2.2.exe)
2. Run the installer executable and accept the license agreement



More info: `http://code.google.com/p/pythonxy/`

Installing on Windows – Python(x,y)

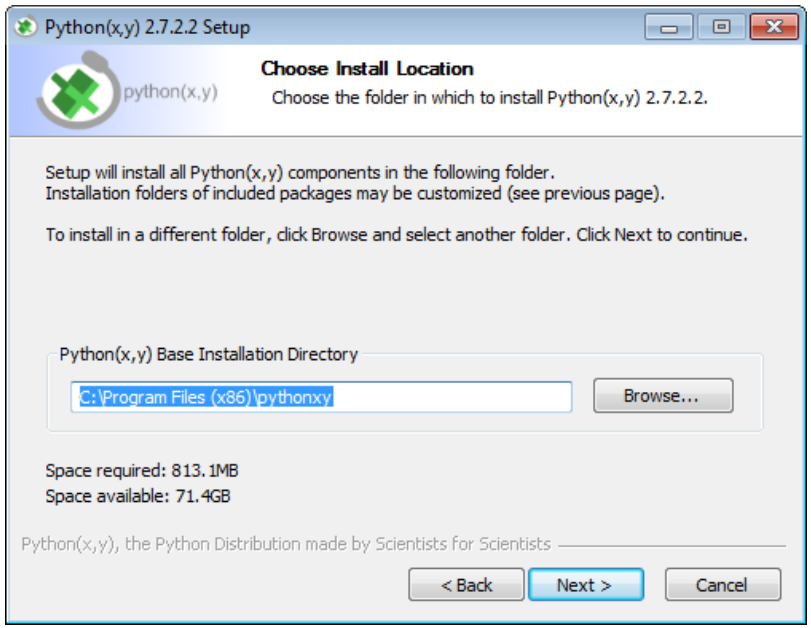
3. Select the components to install

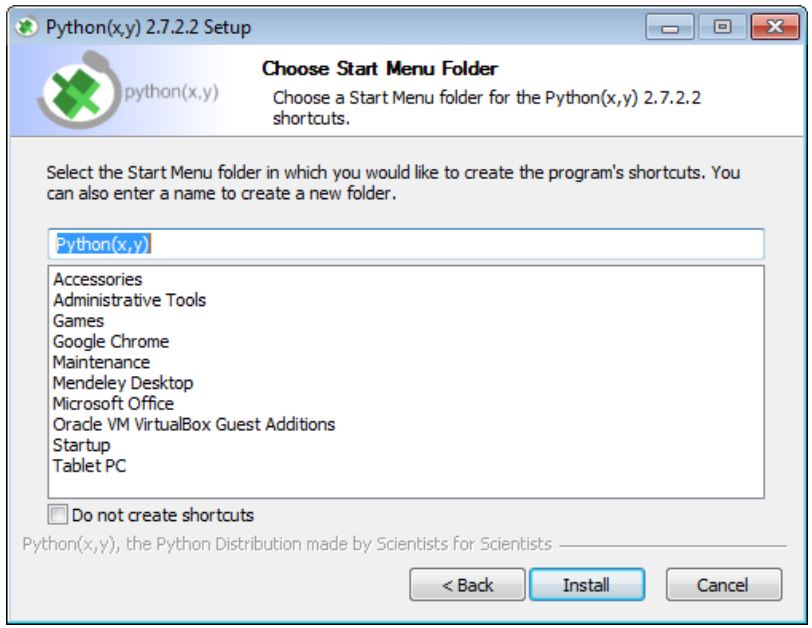


More info: `http://code.google.com/p/pythonxy/`

Installing on Windows – Python(x,y)

4. Select the installation location
5. Select the Start Menu folder
6. Press the **Install** button.

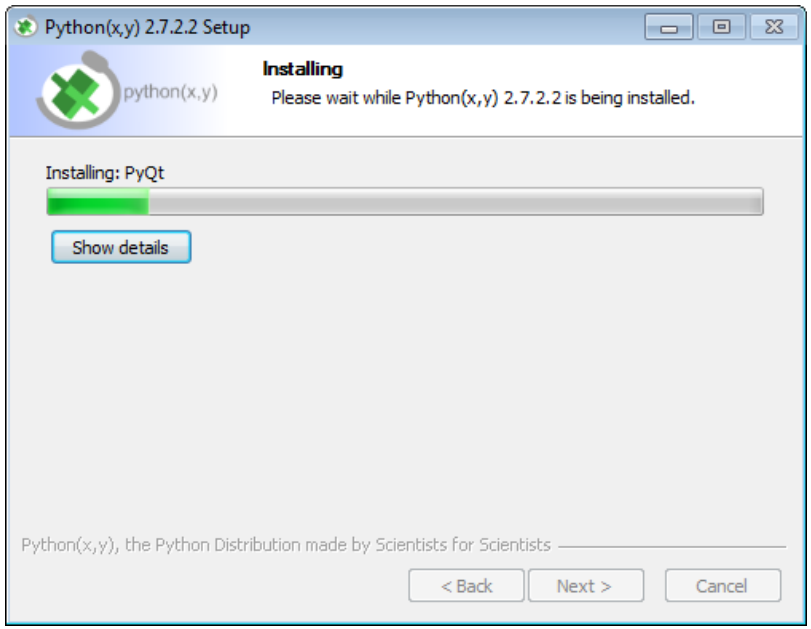


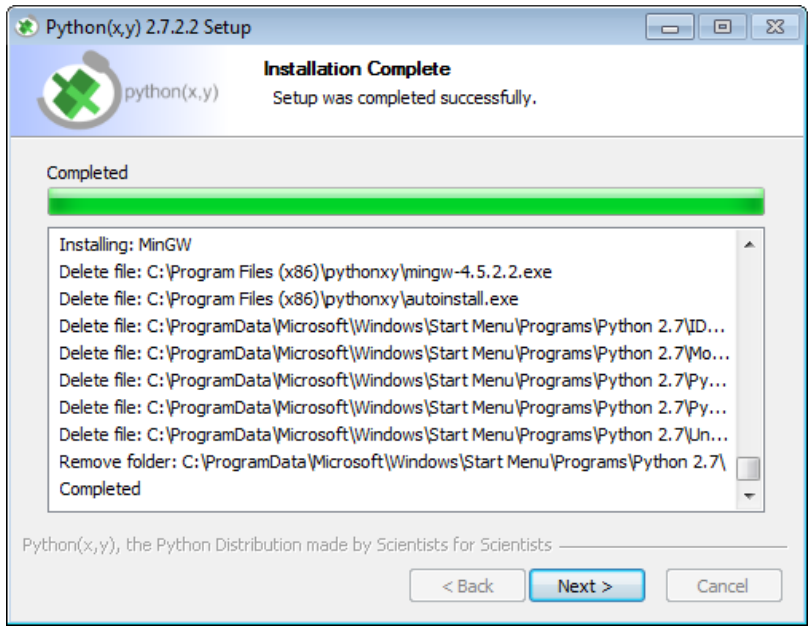


More info: `http://code.google.com/p/pythonxy/`

Installing on Windows – Python(x,y)

7. Wait for everything to install.

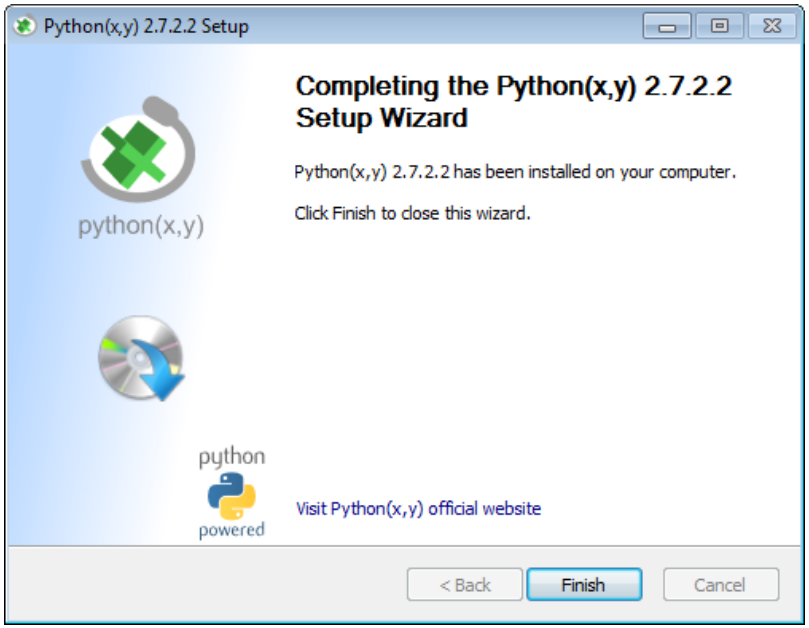




More info: `http://code.google.com/p/pythonxy/`

Installing on Windows – Python(x,y)

8. Press the **Finish** button and you are done!



More info: `http://code.google.com/p/pythonxy/`

Installing on OS X

This should get you setup with all the tools shown today in as "clean" of a way as possible.

Installing on OS X

- The good news:
  - OS X ships with Python installed
  - Homebrew is a great package manager for OS X (`http://mxcl.github.com/homebrew/`).
- The bad news:
  - We need to install xcode
  - Building everything can take a while

Installing on OS X

- First, we assume you have XCode installed.
- Next, install homebrew using the instructions here: `https://github.com/mxcl/homebrew/wiki/installation`.
- Now that we have homebrew, we will install our binary dependencies

```
brew install gfortran # for scipy
brew install pyqt
brew install pyside
brew install opencv
```

- Now we're ready to setup the python stuff in a virtual environment.

Installing on OS X

- What is a virtual environment?
- Python virtual environments let you keep all of your dependencies in one place. (through environment variable magic)
- To install virtualenv, use the instruction here:
  `http://pypi.python.org/pypi/virtualenv`
- Now create a virtual environment like so:

```
virtualenv down_with_matlab
source down_with_matlab/bin/activate
```

- Now we are ready to install our python packages and applications

Installing on OS X

- The order and distinction between pip and easy_install is important.

```
easy_install readline
pip install sphinx
pip install spyder
pip install ipython
pip install tornado
easy_install zmq
pip install numpy
pip install scipy
pip install matplotlib
pip install sympy
easy_install SimpleITK
```

- Now you can use spyder, ipython, and all of these libraries as long as the virtual environment is activated.
- Bonus: None of this is cluttering up your systems site-packages.

Installing on Linux

- The same virtualenv, pip, and easy_install tools used for Mac OS X can be used for Linux, but there is an easier way.
- Most linux distributions have packages for scientific Python tools that make them easy to install.
  - Package are (usually) pre-built, so you don't have to wait for them to compile.
  - Dependencies are automatically installed. If you install Spyder, you automatically get Python, NumPy, SciPy, etc.
- A few toolkits may not be packaged and will still require pip or easy_install.

Installing on Linux - Ubuntu

- In Ubuntu, you can use the Ubuntu Software Center  to select and install packages.
- Equivalently, from the command line you can run

```
sudo apt-get install spyder
```

- Installing Spyder provides most of what is needed.
- Other useful packages to install are

```
sudo apt-get install ipython-notebook ipython-qtconsole python-opencv imagej
sudo easy_install SimpleItk
```

- The apt-get commands will also work on other Debian-based OSes (like Debian and Mint).