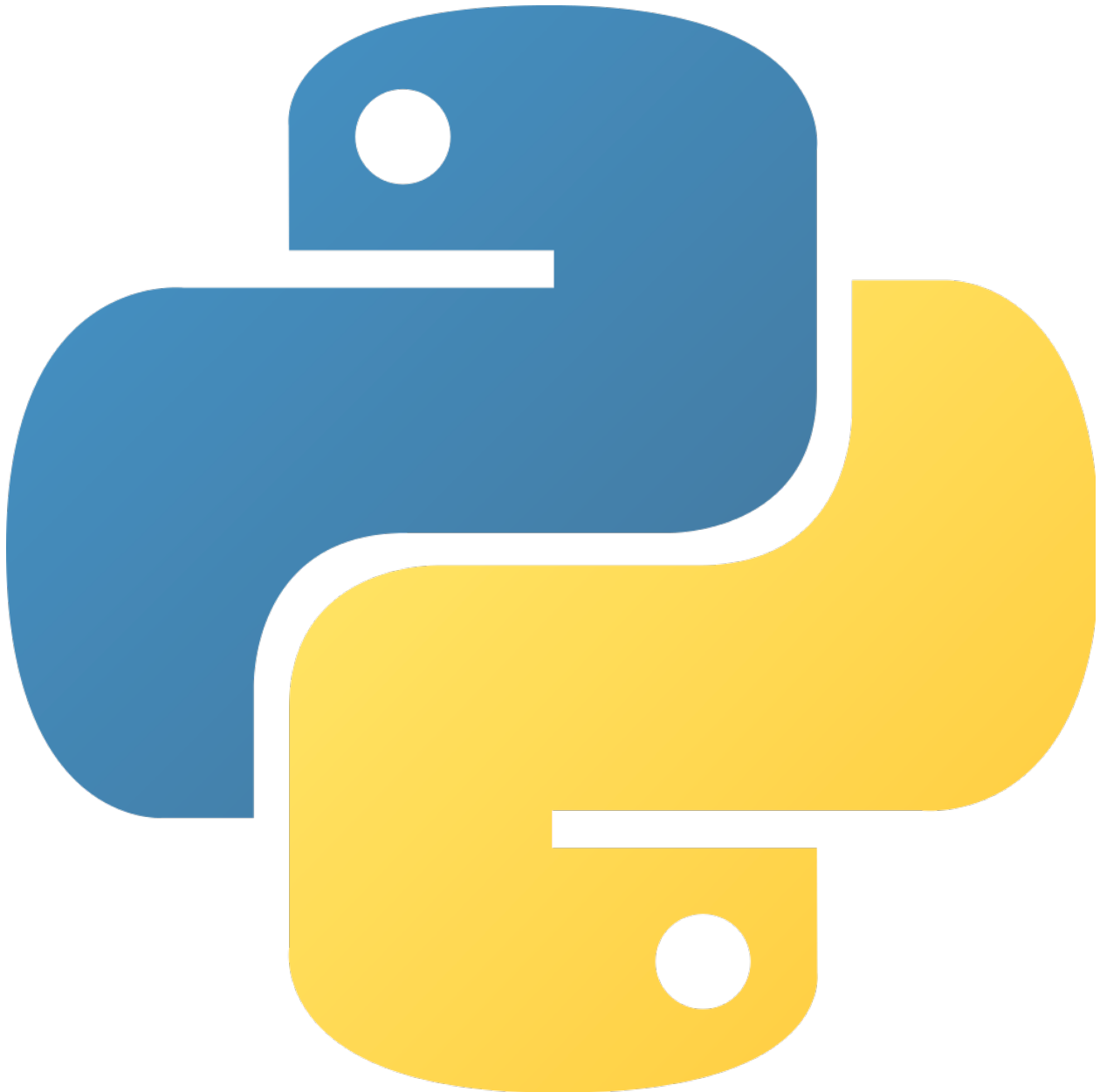


# **Python for Scientific Computing**

## **Quick Start** (Rev 1.2)



Samuel Garcia  
30 October 2016

## Contents

Installing and Setting up Pyzo for Scientific Computing .....	4
Using this Quick Start Guide .....	9
Import Numpy .....	9
Creating Arrays.....	9
Alternative Ways to Create Arrays .....	10
Array Indexing (Slicing) .....	10
Defaults for Indexing .....	10
Negative Indexing and Stepping.....	10
Some Handy Indexing to Keep in Mind .....	11
Array Views / Copy of Array .....	11
Boolean Indexing .....	11
Operators for Boolean Indexing .....	12
Array Data Types and Conversions .....	12
Array Properties .....	12
Assigning Variables.....	13
Array Manipulation .....	13
Reshape an Array.....	13
Flatten an Array.....	13
Tile an Array .....	13
Rearrange Elements .....	14
Transpose of a 2-D Array (or Higher) .....	14
1-D Row Vector into 2-D Column Vector .....	14
Extracting a Column Vector from a 2-D Array.....	15
Concatenation .....	15
Basic Mathematical Operations with Arrays .....	16
Other Mathematical Operations with Arrays .....	16
Dot Product and Matrix Multiplication .....	17
Using the Editor for Script files.....	17
Making Comments .....	17
Running Code within Cells .....	18

Displaying Output .....	18
Advanced Output w/ String Formating .....	19
Blocks of Code: Functions and Flow Control .....	20
Indentation Defines Blocks of Code.....	20
Defining Functions .....	20
Conditions: if, elif, else .....	21
For Loop .....	21
While Loop .....	22
Modules and Packages .....	22
Add Working Directory to the Python Path .....	22
Importing Modules and Packages .....	23
Variations for Importing Modules / Packages .....	24
Importing Numpy (Revisited).....	24
Plotting Graphs: Matplotlib and Pyplot.....	25
Color Options and Better Layout.....	26
Scipy Package.....	26
Example: Signal Processing Module .....	26
Numpy vs Scipy .....	27
Timing .....	27
Reading and Writing Image Files .....	28
Typing Location of File .....	29
imshow() and the cmap parameter.....	30
Changing the Default Colormap.....	30
imsave() - Troubleshooting.....	31
Installing Other Packages .....	31
Future Goals.....	32
References .....	32

## Installing and Setting up Pyzo for Scientific Computing

We have to do three steps:

- 1/ Install a Python interpreter (miniconda)
- 2/ Download and install Pyzo
- 3/ Install any necessary Scientific packages within Pyzo

### Step 1 - Installing the miniconda Python environment

We first need to download a Python interpreter.

We will use miniconda because it's a small download and pretty fast to install.

Go the website: <http://conda.pydata.org/miniconda.html>

Most likely you'll want the **64-bit version** of Miniconda (unless you have a really old computer). Also we'll be using the Python 3.x version (not the 2.x version which is becoming obsolete). As of this writing the latest version is Python 3.5.

Windows is a straight forward .exe installer whereas Mac and Linux uses a bash installer. Bellow we will explain how to install Miniconda on a Mac. If you're a Linux user, the process should be very similar.

### Step 1b - How to install Miniconda on a Mac

We're going to open a terminal window and use three simple unix commands (*cd*, *ls*, and *bash*).

To open a terminal window go to:

**Applications -> Utilities -> Terminal**

In the terminal we now need to change the current directory (i.e. folder) to the where the bash file was downloaded.

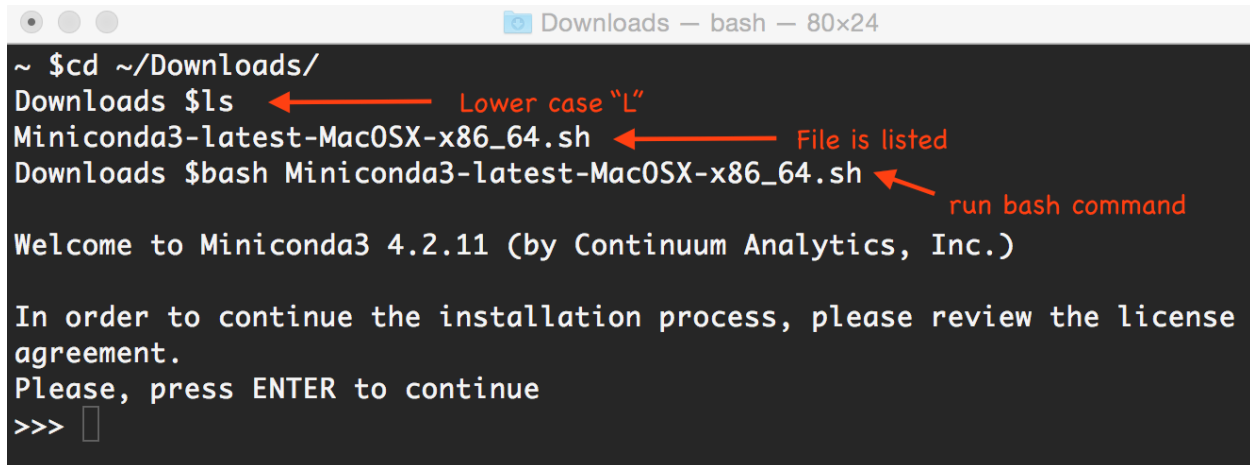
For example, let's say that the bash file was downloaded to your "Downloads" folder in your home folder.

To change to that directory in the terminal we would type:

```
cd ~/Documents
```

To verify that we're actually in the correct directory we'll list all the files in the current directory by typing the command *ls* and verify that the bash file is indeed listed in the current directory of the Terminal window.

Finally we run the bash installer by typing in:  
bash Miniconda3-latest-MacOSX-x86\_64.sh  
At this point your terminal window should look something like the figure below.



```
~ $cd ~/Downloads/  
Downloads $ls  
Miniconda3-latest-MacOSX-x86_64.sh  
Downloads $bash Miniconda3-latest-MacOSX-x86_64.sh  
  
Welcome to Miniconda3 4.2.11 (by Continuum Analytics, Inc.)  
  
In order to continue the installation process, please review the license  
agreement.  
Please, press ENTER to continue  
>>> 
```

Just keep pressing enter to get through the license agreement and follow the directions in the terminal until the bash installer is finished.

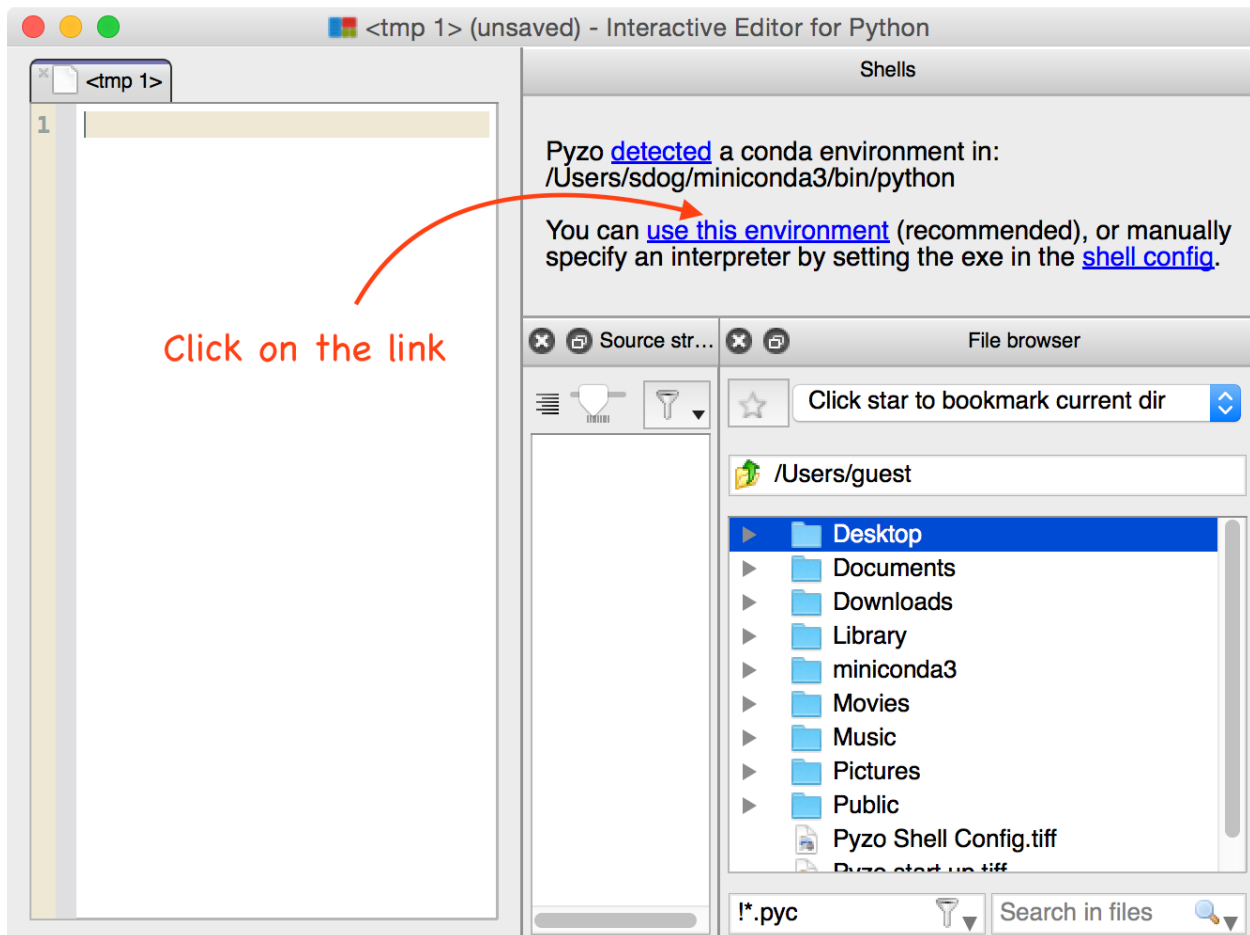
Towards the end, note the location where the Miniconda is installed.

## Step 2 - Download and Install Pyzo

The next step is to download and install Pyzo which is a free and user-friendly Interactive Development Environment (IDE) that runs the Python interpreter which we installed in the previous step. As of this writing the current version of Pyzo is ver 4.3.1. The download file corresponding to your operating system can be found on:

<http://www.pyzo.org/start.html> OR  
at <http://www.pyzo.org/>, and clicking on the "quick start" link

Once we open up Pyzo for the first time we should see something like the figure below:



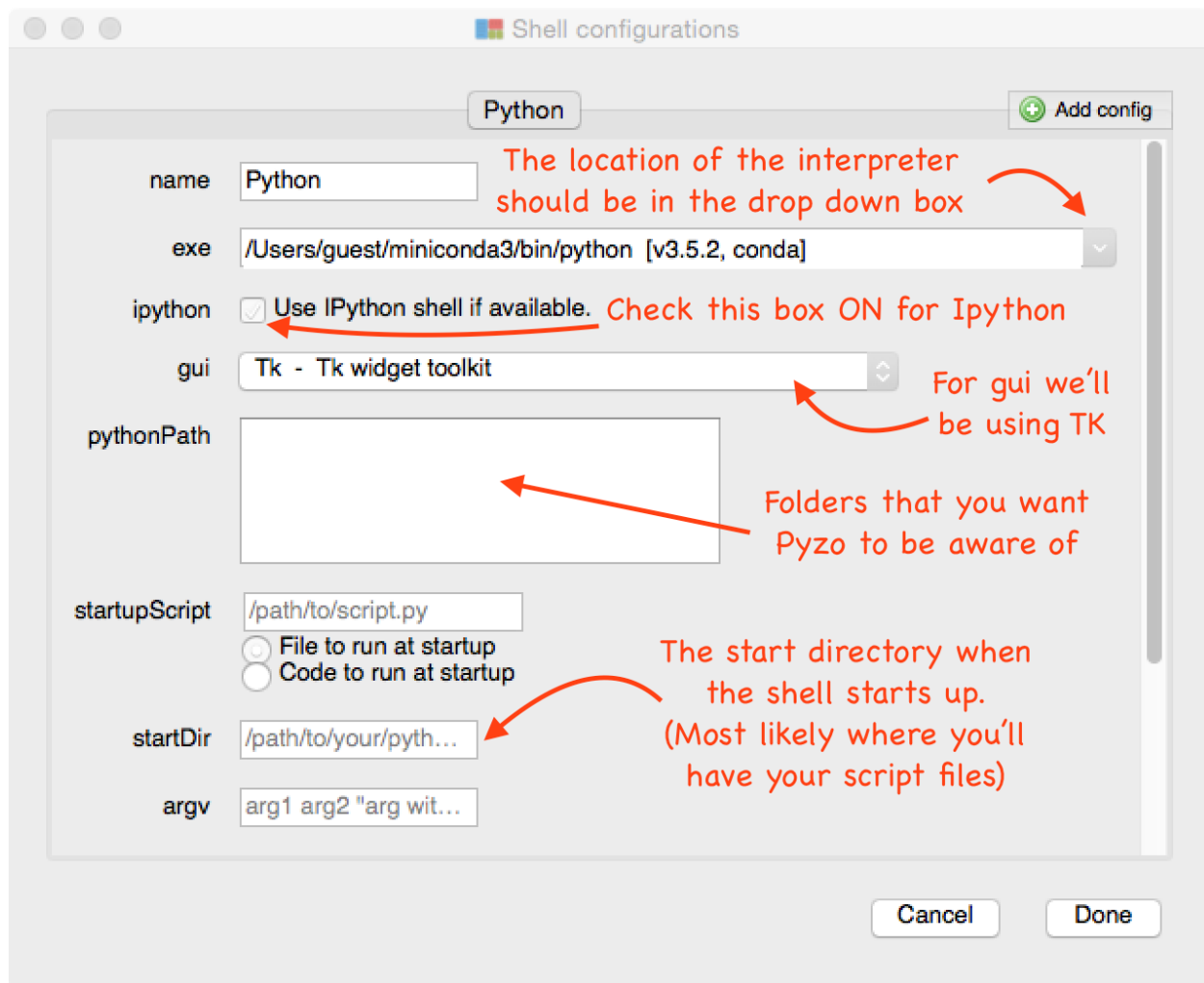
(Note: Later feel free to move around the different window panes)

As indicated on the figure above, click on the link "*use this environment*" to point Pyzo to the location of the Interpreter.

If for some reason, Pyzo can't find the Interpreter, it is remedied in the next step as we have to set up the shell configuration anyway.

In the Menu bar go to **Shell -> Edit Shell configurations**

We should get a window that looks like the figure below. Match up the shell configuration as indicated in the figure notes.



The "startDir" is what you'll use for your default working directory when the Python shell starts up. This is probably where you'll put all your script files. If you're coming from a Matlab environment, Python scripts are analogous to m-files in Matlab.

On the other hand, the pythonPath box is used to specify all the other directories you want Pyzo to be aware of. This is analogous to the search path in the Matlab environment.

The directory paths you use for the "startDir" and "pythonPath" will look slightly different for Windows vs Mac and Linux computers.

Mac and Linux users will have paths similar to:  
/Users/guest/Documents/my\_Python\_Scripts

Whereas Windows users will have paths similar to:  
C:\Users\guest\Documents\my\_Python\_Scripts

Once you are done with the shell configuration, click the “Done” button on the right corner.

### Step 3 - Additional packages to install

The last step is to install some packages that we’ll need to do scientific computing. Here’s the list of packages with a brief description of what they are used for.

**ipython** - for a slightly nicer shell experience

**numpy** - package for using arrays

**scipy** - a scientific package that has most of the stuff we need

**matplotlib** - plotting package

**imageio** - package for reading and writing image files

In the Pyzo IDE, locate the Python shell which should be in the upper right corner (assuming you didn’t move the window panes around yet). The Python shell is analogous to the command window in Matlab.

We will install the packages, one at a time, by going to the shell window pane and typing `pip install` followed by the name of the package we want to install. For example, to install `ipython`, in the shell we would type:

```
pip install ipython
```

Just do this for all five packages listed. After completing this, you can confirm that the packages have been installed typing `conda list`, and you should see the packages we just installed listed on the shell window.

Now let’s restart the shell by going to the menu bar: **Shell -> Restart**, or by using a keyboard shortcut (`cmd + K` in Mac or `ctrl + K` in windows/linux).

Note for Mac Retina Users:

If you’re using one of those fancy retina Mac computers, Pyzo may look kind of fuzzy. To remedy this you can use a program called Retinizer:

<http://retinizer.mikelpm.com/>



## Using this Quick Start Guide

In Pyzo, command statements can be typed directly into the *shell* or typed into the *editor* and then executed.

In order to save space, the following conventions are used in this guide:  
The arrow symbol “->” is used to indicate what will be *stored in memory* after running indicated command statements.

The “>>>” symbol is used to indicate what will be *displayed in the shell* after executing a command.

The hash symbol “#” is used in Python for *making comments* on a line and is used throughout this guide for annotating code.

## Import Numpy

To use arrays in Python we first need to import functions from the Numpy package.

We can do this by typing the following line:

```
from numpy import *
```

The \* symbol means import everything from the module/package.  
Later in this guide we discuss more regarding importing modules.

## Creating Arrays

The easiest way to create arrays is to use the `r_` function:

```
x = r_[23, 4, 1, 100, 42] # array created from a
                           list of numbers
x = r_[1:10:0.2]          # start, stop, step.
                           default step is 1
x = r_[1:10:100j]         # start, stop, num points
# See also linspace() & logspace() functions
```

We can also create arrays using the `ones()` and `zeros()` functions:

```
x = ones(3)    ->  array([1., 1., 1.])
x = zeros(4)   ->  array([0., 0., 0., 0.] )
```

### Alternative Ways to Create Arrays

We can also use the `array()` and `arange()` functions to create arrays

Examples:

```
x = array([1,5,23,2])    # list of numbers
                        same as r_[1,5,23,2]
x = arange(1,10,0.2)     # start, stop, step
                        same as r_[1:10:0.2]
```

### **Array Indexing (Slicing)**

The first element of an array is index 0, not 1

Example:

```
x = r_[23, 4, 1, 100, 42, 66, 9] # example array

x[0] -> 23      # first element is index 0
```

There are many advantages to using zero-based indexing <sup>[2]</sup> which is why it was selected for the Python language.

When indexing an array through slicing, the last element is omitted.

Hence, `x[n1:n1+n]` returns `n` elements

Examples:

```
x[3:5] -> array([100, 42])    # returns x[3] and x[4]
x[3] -> 100
x[0:4:1] -> array([23, 4, 1, 100]) # step size = 1
x[0:4:2] -> array([23, 1])      # step size = 2
```

### Defaults for Indexing

Start=0, End = last element, step = 1

Examples:

```
x[0:5:1] == x[:5:] == x[:5] # first five elements of array
x[::2]                      # every other element
```

### Negative Indexing and Stepping

Negative indices are used to access an array from the end

Examples:

```
x = r_[23, 4, 1, 100, 42, 66, 9] # same sample array as before
x[-1] -> 9
x[-2] -> 66
```

```
x[-4:-2] -> array([100, 42])
```

Negative stepping is used to get array elements in reverse order

Examples:

```
x[-1:-4:-1] -> array([ 9, 66, 42])
```

```
x[::-1]      # elements of array in reverse order
```

### Some Handy Indexing to Keep in Mind

```
x[0]      -> first element
```

```
x[-1]     -> last element
```

```
x[:n]     -> first n elements
```

```
x[-n:]    -> from element -n to the end
```

```
x[::-1]   -> all the elements in reverse order
```

### **Array Views / Copy of Array**

A sliced array is a view of the original array

Example:

```
x = ones([5,5])      # create sample array
y = x[0:2,0:3]       # rows 0 to 1, and columns 0 to 2 of array x
x[0,0] = 5           # modify x at position 0,0
y ->                 # notice that y has been modified also
    array([[ 100.,    1.,    1.],
           [   1.,    1.,    1.]])
```

If we want a separate array, we need to use the `copy()` function

```
y = x[0:2,0:3].copy()
```

In this manner if we later modify `x`, the contents of `y` will not change.

Most of the time we don't need to use `copy`, but it's something important to keep in mind.

### **Boolean Indexing**

In addition to indexing arrays through slicing, arrays can be indexed through boolean indexing.

Examples:

```
x = r_[0:20]          # sample array
idx = (x<5) | (x>=17)  # index of values <4 or >=17
x[idx]                # return values of x at those indices
x[x<3] = 0            # elements less than 3 are set to zero
```

Note: Boolean indexing returns a new array (not a view)

### Operators for Boolean Indexing

AND: &, OR: |, XOR: ^, Negation: ~, Equality: ==, Difference: !=

## **Array Data Types and Conversions**

Numpy supports a lot more data types than Python by itself <sup>[3]</sup>.

The most common data types used are bool, int, float, and complex (the last three are also known as int64, float64, and complex128).

To observe the data type of an array we can use the dtype function.

Example:

```
x = r_[0:5]                # create an array x
x.dtype -> dtype('int64')   # data type of array x
```

If we want to change the data type of an array we can use the astype() function or we can use the data type itself as a function.

Examples:

```
x = x.astype('float')      # change data type from int to float
x = float_(x)              # same as x.astype('float')
                           # note: float_ is equivalent to float64
```

Also when we create an array with a mix of data types, all the elements are converted to the higher data type.

Example:

```
y = r_[10, 11, 12, 3.14159] # array created w/ int and float
y.dtype -> dtype('float64')  # data type is float
```

See online Numpy User Guide <sup>[3]</sup> for more information on data types.

## **Array Properties**

```
size(x) or x.size      # similar to length function in Matlab
shape(x) or x.shape    # similar to size function in Matlab
ndim(x) or x.ndim      # number of dimensions of array
```

## Assigning Variables

Some efficient ways to assign variables:

```
x = array([45, 21, 7, 3, 5]) # sample array
a, b, c = x[0:3]             # multiple variable assignment
a,b = b,a                    # swapping content of variables
```

## Array Manipulation

### Reshape an Array

```
reshape(x,[r,c])             # reshape into 'r' rows and 'c' columns
x.reshape([r,c])             # alternate way to do same thing
```

The default is to populate the reshaped array along the rows and then down the columns, like reading a book.

If we want to populate the reshaped array down the columns and then across the rows, we can use the order parameter:

```
reshape(x,[4,5], order=1)    # fill down the columns
x.reshape([4,5], order=1)    # alternate way to do same thing
```

If we only want to specify one of the dimensions, we can set the other dimension to -1 and Python will figure out the size of the other dimension. Examples:

```
reshape(x,[4,-1])            # Since array has 20 elements, Python knows
                              # to make the unspecified column dimension = 5
x.reshape([-1,5])            # Python makes unspecified row dimension =4
```

### Flatten an Array

```
x.flatten()                  # flatten a 2-D array into a 1-D array,
                              # default is to extract elements along rows

x.flatten(1)                 # flatten 2-D array into 1-D array
                              # but extract elements down the columns
```

### Tile an Array

```
tile(x,[r,c])                # tile an array r by c times
```

Example:

```
x = r_[0:3]          # create an array
tile(x,[2,3])        # replicate it 2 rows by 3 columns
```

### Rearrange Elements

```
flipud(x)            # flip array in up/down direction
                      # if array is 1-D, reverses order of array
fliplr(x)            # flip array in left/right direction
                      # (only works for 2-D array)
```

### Transpose of a 2-D Array (or Higher)

```
x.T    or    transpose(x)    # similar to x.' in Matlab
```

Taking the transpose of 1-D array will not change anything. See next subsection for further discussion.

### 1-D Row Vector into 2-D Column Vector

Most of the arrays we have been creating so far are 1-D arrays.

For example:

```
x = r_[23, 4, 1, 100, 42, 66, 9]
x.ndim -> 1
```

This is different compared with Matlab which always creates a 2-D array.

Taking the transpose of our 1-D array, gives us the same 1-D array.

If we want to make our "row" vector (array) into a "column" vector we can do this in several ways:

```
c_[x]                # useful function (further details in
                      # concatenation section below)
transpose([x])        # add square brackets before taking transpose
array([x]).T          # equivalent to transpose([x])
reshape(x,[-1,1])     # the "-1" is unspecified number of rows
vstack(x)             # vertical stack function
row_stack(x)          # row_stack function
```

### Extracting a Column Vector from a 2-D Array

```
A = reshape(r_[0:20], [4,5]) # create a 2-D array
A[:,2:3]                      # extract all rows from column 2
A[:,[2]]                      # same result
A[:, 2].reshape([-1,1])      # extract 1-D array from 2nd column
                              # and reshape into a 2-D column vector
```

### **Concatenation**

#### For a 1-D Array

The `r_` function can be used for concatenation of a 1-D array.

Example:

```
x1 = array([1,2,3])          # sample array #1
x2 = array([10,11,12])       # sample array #2
y = r_[x1,x2]                # concatenate both arrays
y -> array([1, 2, 3, 10, 11, 12])
```

#### For a 2-D Array

`r_` (for concatenating down the rows) or

`c_` (for concatenating across the columns)

Examples:

```
x = reshape(r_[0:12], [3,4]) # create 2-D array
                              # (3 rows by 4 columns)

r_[x,x]                      # stack x below itself
c_[x,x]                      # stack x to the right of itself
```

If we have a 1-D array and we want to stack down the rows we can do this by adding square brackets.

Example:

```
a = array([1,2,3])          # create 1-D array

r_[a, a]                    # gives us a longer 1-D array
r_[[a,a]]                   # stacks our 1-D arrays on top of
or r_[[a],[a]]              # each other (down the rows)
```

Repeat the three lines of sample code above but with the `c_` function instead and observe the shape of the resulting arrays.

### Other functions that can be used for concatenation

`concatenate()`, `append()`, `hstack()`, `vstack()`

See online Numpy Reference <sup>[4]</sup> for more array manipulation routines.

## Basic Mathematical Operations with Arrays

Addition, subtraction, multiplication, division, and exponentiation with arrays are element by element operations [5].

Examples:

```
x = array([1, 2, 3])      # Sample array #1
y = array([2, 2, 2])      # sample array #2

x*y -> array([2, 4, 6])   # element by element multiplication

x**y -> array([1, 4, 9])   # exponentiation is with ** symbol
                           # the ^ symbol is the XOR function
```

Scalars broadcast (automatically extend) when doing array operations.

Examples:

```
2/x -> array([ 2., 1., 0.666]) # division w/ scalar
2**x -> array([2, 4, 8])        # exponentiation w/ scalar
5+x -> array([6, 7, 8])        # addition w/ scalar
```

It is also easy to do element-by-element operations when using a 1-D array and a 2-D array. Python broadcasts the 1-D array to match the size of the 2-D array.

Examples:

```
A = np.reshape(r_[0:9],[3,3], order = 1) # create 2-D array

x*A                                     # each element of array x gets
                                     multiplied to each column of A
c_[x] * A                             # make x a column vector
                                     and then each element of array x
                                     gets multiplied to each row of A
```

## Other Mathematical Operations with Arrays

Mathematical functions on arrays are applied to each element.

Examples:

```
sin(), cos(), tan(), mean(), std(), median(), sum(), log(),
exp(), ceil(), etc.
```

Some functions can be applied along a single dimension.

Examples:



```

sum(A)          # sum of all elements of 2-D array A
                  (same array from example above)
sum(A,0)        # returns array which is sum of each column
sum(A,1)        # returns array which is sum of each row

```

## Dot Product and Matrix Multiplication

Starting from Python 3.5, the @ operator computes the inner product (dot product) of vectors if the inputs are 1-D arrays, or matrix multiplication if the inputs are 2-D arrays.

Example:

### Inner Product of two vectors

```

x = array([3,4])      # simple 1-D array
x @ x                 # inner product

```

### Matrix Multiplication

```

A = r_[0:6].reshape([2,3])  # create 2-D array (2 by 3)
B = c_[[1,1,1], [2,2,2]]    # another 2-D array (3 by 2)

AB = A @ B                  # result is 2 by 2 array

```

For other useful matrix and linear algebra functions, such as matrix determinant, inversion, eigenvalues, etc., see the online documentation for the Numpy Linear Algebra routines<sup>[6]</sup>.

## Using the Editor for Script files

For short snippets of code, typing commands directly into the shell is feasible. However, for longer pieces code it is much more convenient to use the editor and create script files. Script files use the .py extension, analogous to Matlab m-files using the .m extension.

### Making Comments

As demonstrated throughout this tutorial, the hash symbol “#” is used to make comments on a line. Anything after the “#” is ignored by the interpreter until the next line.

We can also comment multiple lines by using triple quotes <sup>[7]</sup>. This is especially useful for commenting-out long portions of code for debugging.

Example:

```
'''
This is a multiline
comment.
'''
```

Note: Triple quotes are also used for something called *docstrings* <sup>[7][8]</sup>.

### Running Code within Cells

In Pyzo, we can either run the entire code in the script file, or just code within a particular cell. A cell is recognized as everything between two lines starting with two hash symbols “##”.

Cells are a unique feature of Pyzo (in comparison to other Python IDEs) and makes it easy to separate a script file into sections. Using the Source Structure window pane of Pyzo we can also easily observe all the cells of a script file.

### Displaying Output

In Python there is no need for any syntax to suppress displaying output to the shell. (Compare this with Matlab which always needs a semicolon to suppress output). Instead, if we want to display output to the Python shell we need to use the `print()` command.

Example:

```
x = r_[1:6]      # sample array
y = x+10         # y is a function of x
print(y)         # display y to the shell
>>> [11 12 13 14 15]
```

We could also use text to display a more descriptive output.

Note the use of the comma.

Examples:

```
print("y is ", y)      # display everything on same line
or
print("y is \n", y)    # the "\n" indicates a new line
```

### Advanced Output w/ String Formating

Working with strings, we can display more advanced outputs by using the `.format()` function which was introduced in Python version 2.6.

Example:

```
# store a formatted string to the variable str1
# and print the formatted string

H,L,W = 5, 3, 4      # length, width, and height dimensions
str1 = "The box is {1} x {2} x {0} cm".format(H,L,W)
print(str1)
>>> The box is 3 x 4 x 5 cm
```

Notice that the curly braces `{}` are place holders. The numbers inside the place holders are indices which determine which argument in the format function gets substituted into the place holder. Hence, in the example `{0}` substitutes the height `H`.

The place holders can also have a format specification which modifies things such as how many decimal places to display for floating point numbers or displaying numbers with a comma separator. The format specification is indicated after the colon symbol `:"`.

Example:

```
x = 3.141592653589793      # store value of pi to variable
str2 = "Pi is {0:.4f} to four decimal places".format(x)
print(str2)
>>> Pi is 3.1416 to four decimal places
```

The `“.4` refers to four decimal places and the `“f”` refers to floating point.

Instead of using numbered indices in the placeholders, we can also use keyword arguments.

Example:

```
x,y = 'male', 150
str3 = "Person is a {w} lb {sex}".format(sex=x, w=y)
print(str3)
>>> Person is a 150 lb male
```

Lastly, place holders can be embedded within placeholders.

Below is a more elaborate example that uses a place holder for the number of decimal places to display.

```
gr = 1.61803398875          # Golden Ratio
n = 5                      # variable number of decimal places
str4 = "Ratio is {x:.{dig}f} to {dig} digits".format(x=gr,dig=n)
print(str4)
>>> Ratio is 1.61803 to 5 digits
```

When in doubt, just remember that brackets are simply placeholders.  
For more details regarding using strings in Python see references <sup>[9]</sup><sup>[10]</sup><sup>[11]</sup>.

## **Blocks of Code: Functions and Flow Control**

### Indentation Defines Blocks of Code

Defining blocks of code in some languages requires brackets (i.e. C, C++) or a keyword such as end (i.e. Matlab).

Instead, Python uses indentation to define blocks of code. The advantage is that code is easy to read and the user can code with less typing.

The convention in Python is to use 4 spaces for indentation. Although it is recommended to use this convention, tabs and other space sizes will also work as long as its consistent throughout the same script file.  
(i.e. don't mix tabs and spaces in the same file)

### Defining Functions

Unlike Matlab, which requires each function to be saved in its own separate file, functions in Python can be defined anywhere in a script file.

Below is an example of a user-defined function.

(Note the use of indentation to define the code for the function)

```
def divide_then_add(a, then_add=0):
    y = a/2                # divide by 2
    y = y + then_add        # add value of then_add
    return y               # output of function
```

Functions are always defined with the *def* Python keyword followed by the name of the function, a set of parenthesis "()", and a colon symbol ":".

Inside the parenthesis we can have any arbitrary number of arguments (inputs) to the function. We can also make arguments into keyword arguments with default values. (In our example we made *then\_add* into a keyword argument with a default value of 0).

Lastly, the return Python keyword is used to return one variable (of any type).

Let's see how we call our user-defined function.

```
x = r_[0:4]           # create an array
divide_then_add(x)    # calling our function
>> [0. 0.5 1. 1.5]
divide_then_add(x,3)  # same as divide_then_add(x,then_add=3)
>> [3. 3.5 4. 4.5]
```

### Conditions: if, elif, else

Python uses boolean expressions to evaluate conditions. Note the use of indentation and the colon symbol.

```
if x<3:
    print("x is small")
elif (x>=3) & (x<7):
    print("x is medium")
else:
    print("x is large")
```

Any combination of if, elif, and else can be used.

Examples:

- if statement alone
- if followed by else statement (no elif)
- if followed by any number of elif statements

### For Loop

The for loop iterates over elements from an array (or in general from any "iterable" such as a list, tuple, or string). Again note the use of indentation and the colon symbol.

Example:

```
cnt = 0                # count num of iterations through loop
for x in r_[15:18]:    # use array as "iterable"
    cnt = cnt + 1
    print(x)
print("num of loops is ", cnt)  # three times through loop
                                because array is length 3
```

The keyword `break` is used to exit a loop, whereas the keyword `continue` is used to loop around (i.e. skip the rest of the statements in the block and go to next iteration of loop).

Example:

```
for x in r_[0:12]:      # iterate over array
    if 5<=x<=8:         # if x between 5 and 8 (inclusive)
        continue        # then skip the rest of the statements
    print(x/2)
```

Try the code above again, but replacing the `continue` keyword with `break` and observe the resulting output.

### While Loop

The `while` loop is used to repeat a block of code as long as a certain boolean condition is met.

Example:

```
x = 10
while x>0:      # repeat loop while x is not negative
    print("x is ", x)
    x = x-1     # decrement x by 1
```

The keywords `break` and `continue`, discussed in the previous section, can also be used with `while` loops. Another noteworthy keyword is `pass` which can be used as a placeholder for the code of a function or a control loop. This can be useful when a user is working on new code.

Example:

```
def my_func():
    pass      # do nothing: will fill-in code at a later time!
```

## **Modules and Packages**

### Add Working Directory to the Python Path

Before continuing make sure your working directories (where you're saving your script files) are included in the Python path.

To do this in Pyzo go to the "Shell" menu and click "Edit shell configurations".

Then in the “pythonPath” box enter in the desired directory paths you would like to add (one per line). Next, restart the shell and your working directories should now be included in the Python path. (keyboard shortcut for restarting the shell is ctrl+k in Windows/ Linux or cmd+k on Mac.)

To see a list of all the directories included in the pythonPath we can type the following commands:

```
import sys                # module w/ system functions
from pprint import pprint as pp  # for printing in nice format
pp(sys.path)              # print python path
```

### Importing Modules and Packages

Modules are just regular Python files with the .py extension that contain definitions of functions, variables, and objects, whereas packages are folders which contain related modules. In a sense Python modules are analogous to Matlab m-files, and Python packages are analogous to Matlab toolboxes.

Let’s say we create a script with various function definitions and save it with the name *myfile.py*. In order to use the functions from that module we need to use the `import` command.

Example:

```
import myfile              # one way of using the import cmd
```

Notice when importing the module that we don’t include the .py extension. We can now use any function defined in the module:

```
myfile.fun1(a,b)          # use function called fun1
myfile.fun2(x)            # use function called fun2
```

The `import` command is used both for importing modules and packages: the syntax is exactly the same. In fact, throughout this tutorial we have been using the Numpy package which we imported in the first page of this guide.

Other packages of interest include Matplotlib and Scipy which we’ll explore later in this guide.

## Variations for Importing Modules / Packages

Since the syntax is the same for both importing a module or a package, in the following explanations we'll just refer to the process of importing a module, knowing that the same explanation applies to packages also.

In the previous section we looked at just one way of importing a module. Using the same example from the section before, if we want to use the function *fun1* from the *myfile* module, we can do this in four different ways. Examples:

```
import myfile          # option 1: same as previous section
myfile.fun1(a,b)       # takes a lot of typing to call function

from myfile import fun1 # option 2: easy to call a function,
fun1(a,b)              # difficult to import many functions

from myfile import *    # option 3: easy to call a function
fun1(a,b)              # but could lead to name collisions

import myfile as my     # option 4: recommended way to import
my.fun1(a,b)           # less typing and no name collisions
```

As can be observed above, each import option has its advantage and disadvantage. Although option 3 is convenient to use it should be avoided as much as possible to avoid name collisions (i.e. different functions from different modules sharing the same name).

In most situations, the recommended way to import a module is to use option 4 as illustrated in the example above. It doesn't require as much typing and avoids the problem of name collisions.

## Importing Numpy (Revisited)

In the beginning of this guide we were unfamiliar with how to import modules. Hence, for ease of use, we resorted to importing the Numpy package using "from numpy import \*" (option 3 from previous section). However, building upon the discussion from the previous section, it is recommended to instead import Numpy in the following way:

```
import numpy as np      # use the letters np to save typing
```



Note that we could have selected any letters that we wanted instead of using "np". However, it is somewhat of a convention to use np when importing Numpy. (A lot of documentation and examples from various online sources tend to use this convention.)

Using this method to import Numpy, all the commands discussed previously are called in exactly the same way, except that now we prepend each command with the letters np. See the following examples:

```
x = np.pi * np.r_[0:20]      # create an array
x = x.reshape([4,5])         # reshape into 4 rows x 5 columns
s = np.sum(x,0)               # sum of each column
```

If we use some functions regularly and want to save typing by not using the full function name, we could save them with a different object name:

```
r_, c_ = np.r_, np.c_        # save functions as r_ and c_
                                # in order to save typing
tp = np.transpose             # shorter name for transpose function
```

## Plotting Graphs: Matplotlib and Pyplot

In order to create plots in Python, we need to import a module called pyplot, which is part of the matplotlib package. Matlab users should find that using pyplot is fairly easy since plotting commands and syntax in pyplot are very similar to those found in Matlab.

Examples:

```
import matplotlib.pyplot as plt    # use plt to save typing
x = np.r_[0:2:0.1] * np.pi        # create array
y = np.cos(x)                      # y is a function of x
plt.plot(x,y,'r')                  # plot x vs y in red
plt.grid(1)                        # grid on
```

Examples of other pyplot functions:

(remember to prepend "plt." before the functions)

```
axis(), axes(), figure(), grid(), legend(), plot(), subplot(),
title(), xlabel(), ylabel(), xlim(), ylim()...
```

If we would like to save the current figure to an external file in the png or pdf format we can use the `savefig()` function.

Example: `plt.savefig("my_fig.png")`

Note: When running commands from the pyplot module you may get a warning message requesting the user install pyparsing version 2.0.0 or newer. In order to get rid of this warning message just type:

```
pip install pyparsing --upgrade
```

This will upgrade pyparsing to the newest version and will get rid of the annoying warning message <sup>[14]</sup>.

### Color Options and Better Layout

When using the plot function from pyplot you can specify various colors. The standard built-in colors can be specified with just a single letter. (Blue, green, red, cyan, magenta, yellow, black, and white are specified with b, g, r, c, m, y, k, and w respectively.) If these standard colors are not enough, we can also specify colors using an html color name.

Examples:

```
plt.plot(x,y1,'k')      # plot curve in black
plt.plot(x,y2,'teal')   # plot curve in teal
```

When using the subplot function of pyplot, sometimes the labels of different subplots may overlap each other. Instead of adjusting this manually using the figure window toolbars, we can take care of this automatically by using the `tight_layout()` function.

Example:

```
plt.tight_layout()      # ensure subplot titles don't overlap
```

For more information on using the pyplot module see references <sup>[13]</sup> to <sup>[16]</sup>.

### **Scipy Package**

The Scipy package is composed of several sub-packages including:

```
scipy.integrate      # for integration and ODE's
scipy.linalg         # for linear algebra routines
scipy.optimize       # for optimization and root finding
scipy.interpolate    # for interpolation
scipy.signal         # for signal processing
scipy.stats          # for statistics
```

For a complete list of included modules in the Scipy package see <sup>[17]</sup>.

### Example: Signal Processing Module

```
import numpy as np      # import numpy to use arrays
import scipy.signal as sp # use abbreviation sp to save typing
```

```

h = [4, 2, 1]           # filter coefficients of FIR filter
x_in = np.r_[0:6]       # create input signal
y = sp.lfilter(h,1.,x_in) # filter the input signal

```

Notice: The filter function has the general form `lfilter(b,a,x)`, where `b` and `a` are the numerator and denominator coefficients of the filter respectively, and `x` is the input signal. In order for the `lfilter()` function to work properly, at least one of those three arguments should be of type float. This is why we used the floating point number "1." as the second argument in the example.

### Numpy vs Scipy

There is some overlap of functions in both Scipy and Numpy. For instance, we can use either `scipy.linalg.inv()` or `numpy.linalg.inv()` for taking the inverse of a matrix. Another example is in taking the Inverse Fourier Transform of an array:

```

import scipy.fftpack as sf    # import fft module from scipy
import numpy.fft as nf       # and also from numpy
sf.ifft(x)                   # take inverse fft in scipy
nf.ifft(x)                   # same function but using numpy

```

In many cases the implementation, functionality, and speed of these functions is very similar in both Scipy and Numpy. However, in some cases a function in Scipy might be a little more advanced than the Numpy version. For example, `numpy.convolve()` does a linear convolution of two 1-dim arrays, whereas `scipy.signal.convolve()` does a linear convolution of two N-dim arrays. In this case the advanced functionality may come at a price of increased processing time.

### **Timing**

To measure the execution time of Python code we can use the time module. Example:

```

import time                # import time module
t1 = time.time()           # record first time in variable t1
'''
INSERT CODE HERE
'''
t2 = time.time()           # record second time in variable t2
print( "Elapsed time is {} sec".format(t2-t1) )

```

For more information on timing in Python see references [18][19].

## Reading and Writing Image Files

In order to read and write image files we can use a package called imageio. For displaying images in figure windows we again use the pyplot module.

```
import imageio as iio          # reading/ writing images
import matplotlib.pyplot as plt # display images in figures
import numpy as np             # for array manipulation
```

In the example below, we read an image file and then display the image along with two of the RGB components in a figure window. We then set one of the RGB components to zero and save the modified image to a new file. Example: (Color added to code to make reading easier)

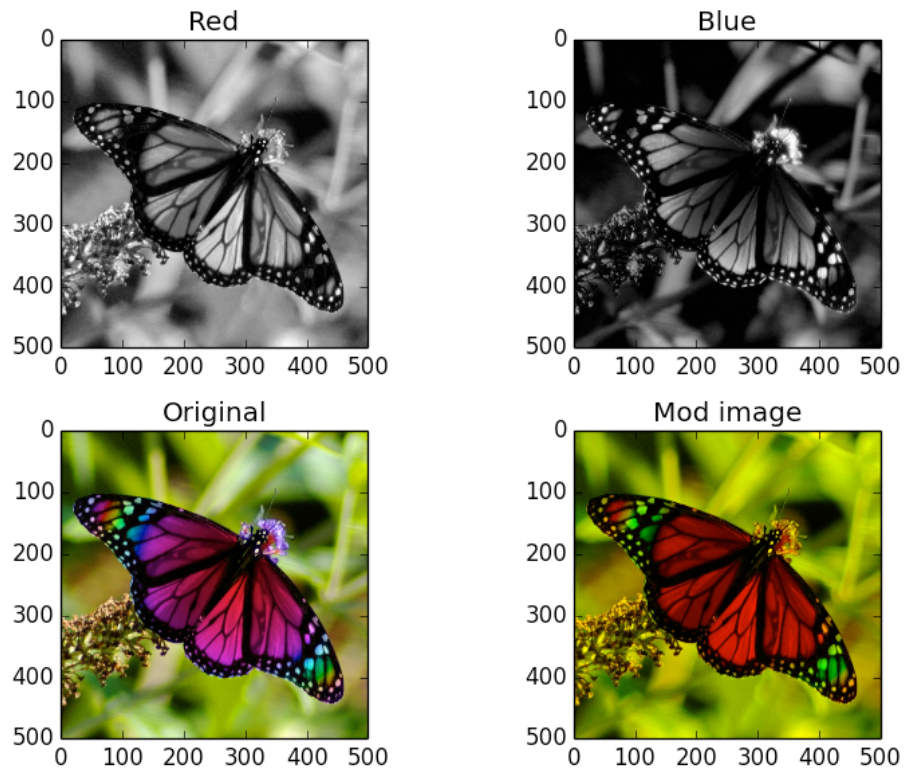
```
# Read image from file
# use variable for directory path of input file (to save typing)
fname1 = "/Users/Anakin/python_scripts/butterfly-rainbow.jpg"
A = iio.imread(fname1)          # Read the image to an array

# Show Images in Figure Window
plt.subplot(2,2,1); plt.title("Red")      # Show Red Component
plt.imshow(A[:, :, 0], cmap="gray");
plt.subplot(2,2,2); plt.title("Blue")    # Show Blue Component
plt.imshow(A[:, :, 2], cmap="gray");
plt.subplot(2,2,3); plt.title("Original") # Orig Image
plt.imshow(A)

# Set the Blue Component to zero
B = A.copy()                        # Copy the image array
r,c,p = B.shape                     # get the dimensions
B[:, :, 2] = np.zeros([r,c])        # set blue component to zero

# Show the modified Image
plt.subplot(2,2,4); plt.imshow(B)
plt.title("Mod image")

# Save the Modified Image
# use variable for path of output file (to save typing)
fname2 = "/Users/Anakin/python_scripts/butterfly-mod.jpg"
iio.imwrite(fname2,B)
```



### Typing Location of File

When using `imread()` and `imsave()` in the previous example, we used variables to save the directory path of the image files in order to save typing. Alternatively, if the files are already in the current directory of the shell, it is not necessary to do this.

Example:

```
iio.imread("butterfly-rainbow.jpg")
iio.imsave("butterfly-mod.jpg")
```

In Pyzo we can show the shell's current directory or change it to another directory by using the command "cd" in the shell prompt.

Example:

```
>> cd                                     # verify cd of shell
'/Users/Anakin/'

>> cd python_stuff                       # change cd of shell
'/Users/Anakin/python_stuff'
```

A more efficient method to change the directory path is to go to the "Shell" menu and click on "Edit shell configurations". Then in the "startdir" box enter in the desired directory path. Now whenever we start up Pyzo or restart the shell, the shell will automatically be in the desired directory path where we want to read and write image files. This can help save us a significant amount of typing.

### imshow() and the cmap parameter

For displaying an image to the figure window we used the `plt.imshow()` function. Notice that we also used the optional color map parameter (`cmap="gray"`) when using this function on single channels of the color image.

To understand why we used this `cmap` parameter, first note that each of the three channels can be considered an individual image of intensities (also called a monochromatic image). Hence, if we extract the R channel of the color image, we can observe the pixels that have greater intensities of red.

Since a monochromatic image only designates intensities, we need to use a color map to associate different intensities with different colors. Using a `cmap` set to `gray` associates low intensities with black, high intensities with white, and everything in between with different shades of gray. There are various `cmap` settings to choose from <sup>[20]</sup>. If no `cmap` parameter is designated, `imshow()` defaults to a `cmap` setting called "jet".

### Changing the Default Colormap

If we use a particular colormap frequently when displaying monochromatic images, it may be more convenient to change the default colormap.

Example:

```
plt.gray()                # change default cmap to gray
plt.imshow(A[:, :, 0])    # display red channel with cmap=gray
```

Other colormaps we can use for the default include:

```
autumn(), bone(), copper(), flag(), gray()...
```

For a complete list of default colormap options see <sup>[21]</sup> and search for "default colormap" on the webpage.

### imsave() - Troubleshooting

If the shell gives you an error when using the `imsave()` function from the `imageio` package, try including the `format` parameter:

```
imageio.imsave("butterfly-mod.jpg", format="jpg")
```

There are many image processing routines in the `Scipy` package (`scipy.ndimage`) and as well as in the `scikit-image` package (`skimage`). For more information regarding Image Processing in Python see <sup>[22][23][24]</sup>.

### **Installing Other Packages**

Pyzo comes with *pip* and *conda*: two package managers that are used to install and manage packages. To see the list of packages that are included with Pyzo we can type:

```
pip freeze          # use pip to list installed packages
conda list          # use conda to list installed packages
```

In a default installation of Pyzo, the list of installed packages will be about the same in `pip` and `conda`, although, there may be one or two packages that are in one but not the other.

If we want to install a package not included in Pyzo, it's best to search the internet for instructions on how to install that particular package.

In many cases a package can be installed with `pip`.

(Example: "`pip install some_package`" where we substitute `some_package` with the actual name of the package.)

For more information on `pip` and `conda` see references <sup>[26][27][28]</sup>

## Future Goals

This document is a work in progress. Python has a very active community, and so as the language and packages evolve over time, this document will attempt to keep up with those changes. Some future sections I plan to add to this guide include:

- Adding a few more functions from the Numpy module (in1d, argmin, argmax, all, any)
- Add section regarding Lists in Python - used as general containers (similar to cells in Matlab)
- Include some useful functions to be used with lists (len, append, extend, count, remove)
- Add section on defining "Anonymous" functions. (i.e. inline functions) and also on how to use this for defining Piecewise Functions
- Add section of how to use Residuez function in Signal Processing module of Scipy
- Add section on working with text and data files.
- Add section on how to create your own Python Package

## References

[0] Python as your next Matlab?, [Online]  
Available: <https://ep2013.europython.eu/media/conference/slides/pythonxy-python-next-matlab.pdf>

[1] IEP Wizard, [Online]  
Available: <http://www.iep-project.org/iepwizard.html>

[2] E.W. Dijkstra, "Why Numbering Should Start at Zero", [Online]  
Available: <https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF>

[3] Numpy User Guide - Data Types, [Online]  
Available: <http://docs.scipy.org/doc/numpy/user/basics.types.html>

[4] Numpy Reference - Array Manipulation Routines, [Online]  
Available: <http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html>

[5] Python Operators, [Online]  
Available: <https://docs.python.org/3/library/operator.html#mapping-operators-to-functions>

[6] Numpy Linear Algebra Routines, [Online]  
Available: <http://docs.scipy.org/doc/numpy/reference/routines.linalg.html>



- [7] Multiline Comments in Python, [Online]  
Available: <http://stackoverflow.com/questions/7696924/multiline-comments-in-python>
- [8] Docstring Conventions, [Online]  
Available: <http://legacy.python.org/dev/peps/pep-0257/>
- [9] How to Think Like a Computer Scientist, Ch 8. Strings [Online]  
Available: <http://openbookproject.net/thinkcs/python/english3e/strings.html>
- [10] Python String Format Cookbook, [Online]  
Available: <http://mkaz.com/2012/10/10/python-string-format/>
- [11] Python 3 Tutorial, Formatted Output, [Online]  
Available: [http://www.python-course.eu/python3\\_formatted\\_output.php](http://www.python-course.eu/python3_formatted_output.php)
- [12] Pyzo Forum, [Online]  
Available: <https://groups.google.com/forum/#!msg/pyzo/Li9xzhBfTY/-kBBYGsACTYJ>
- [13] Pyplot Commands Summary, [Online]  
Available: [http://matplotlib.org/api/pyplot\\_summary.html](http://matplotlib.org/api/pyplot_summary.html)
- [14] Pyplot Tutorial, [Online]  
Available: [http://matplotlib.org/users/pyplot\\_tutorial.html](http://matplotlib.org/users/pyplot_tutorial.html)
- [15] Matplotlib How To FAQ, [Online]  
[http://matplotlib.org/faq/howto\\_faq.html](http://matplotlib.org/faq/howto_faq.html)
- [16] Matplotlib Tight Layout Guide, [Online]  
[http://matplotlib.org/users/tight\\_layout\\_guide.html#plotting-guide-tight-layout](http://matplotlib.org/users/tight_layout_guide.html#plotting-guide-tight-layout)
- [17] Scipy Reference Guide, [Online]  
Available: <http://docs.scipy.org/doc/scipy/reference/>
- [18] Python Standard Library Documentation, Ch 16.3 - Time, [Online]  
Available: <https://docs.python.org/3.3/library/time.html>
- [19] Python Central Website, Time a Python Function, [Online]  
Available: <http://www.pythoncentral.io/time-a-python-function/>
- [20] Matplotlib Colormap Reference, [Online]  
Available: [http://matplotlib.org/examples/color/colormaps\\_reference.html](http://matplotlib.org/examples/color/colormaps_reference.html)
- [21] Matplotlib.pyplot Reference, [Online]  
Available: [http://matplotlib.org/api/pyplot\\_api.html](http://matplotlib.org/api/pyplot_api.html)
- [22] Matplotlib Image Tutorial, [Online]  
Available: [http://matplotlib.org/users/image\\_tutorial.html](http://matplotlib.org/users/image_tutorial.html)

- [23] Image Processing using Scipy and Numpy, [Online]  
Available: [http://scipy-lectures.github.io/advanced/image\\_processing/](http://scipy-lectures.github.io/advanced/image_processing/)
- [24] Scikit - Image Processing, [Online]  
Available: <http://scipy-lectures.github.io/packages/scikit-image/index.html#scikit-image>
- [25] Butterfly Rainbow Image, [Online]  
Available: <http://www.universeofsymbolism.com/images/butterfly-rainbow.jpg>
- [26] Pip Website, [Online]  
Available: <https://pypi.python.org/pypi/pip>
- [27] Conda Website, [Online]  
Available: <http://conda.pydata.org/>
- [28] StackOverflow Website - Difference Between pip and conda, [Online]  
Available: <http://stackoverflow.com/questions/20994716/what-is-the-difference-between-pip-and-conda>
- [29] Python for Scientists - Python vs Matlab, [Online]  
Available: <https://sites.google.com/site/pythonforscientists/python-vs-matlab>