Introduction

This tutorial is structured around the idea that you want to get up and running with Python using `PyLab` as quickly as possible. The first question I asked my myself before I started using the Python *scipy stack* was why consider Python in the first place? What makes it a viable alternative to other languages available for scientific and engineering computations and simulations? OK, everyone has favorites, and presently MATLAB is very popular in the signals and system community. Is there a need to change? This is a debate that lies outside the scope of this tutorial, but the ability to use open-source tools that work really, really well is very compelling.

To answer the first question, why consider Python, I can say:

1. The *NumPy* library.
2. combined with *Matplotlib* .
3. The *SciPy* library of modules, particularly *signal*, provides reasonable suppost for signals and systems work.
4. Additional libraries of modules are also available, in particular `scikit-dsp-comm` at (https://github.com/mwickert/scikit-dsp-comm).

Before Numpy

I have been saying a lot about using Python with Numpy as a means to do scientific and engineering analysis, simulation, and visualization. The fact of the matter is, Python is a good language for doing many other things outside the computational realm.

Numpy plus Scipy are key elements to the attractiveness of using Python, but before getting too carried away with the great scientific computing abiliies of the language, you should learn some basics of the language. This way you will feel more comfortable at coding and debugging.

Before exploring the core language, I will spend time going over the environment and various choices.

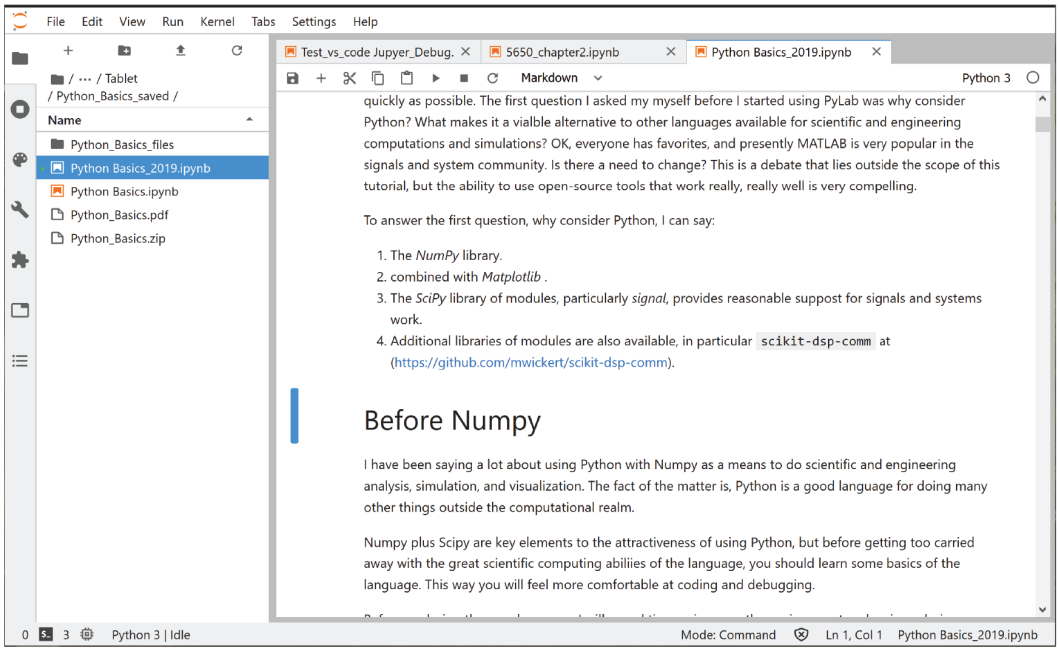
The Environment and Choices

How you choose to work with Python is up to you. I do have some strong suggestions. But first I want to review four options in order of most recommended to least recommended. My recommendations assume you are just starting out with Python, so I have a bias towards the Jupyter notebook, and in particular the use of `jupyter Lab` as opposed to the original `jupyter notebook` .

The first thing you want to do is get a version of Python with scientific support included. When this notebook was first created I was using Canopy, but now my preference is to use Anaconda or miniconda. For beginners the full Anaconda is likely the best starting point as you just have to a few more packages to the base install to be ready to roll with scikit-dsp-comm. With miniconda you have a simple base environment but create from scratch a virtual environment that has just the packages you desire. See for example Create a Python 3.7 Virtual Environment.

Option 1: Launching the Jupyter Notebook

Regardless of the operating system, Windows, Mac OS, or Linux, you want to get a terminal window open. It is best if the terminal window is opened at the top level of your user account, so you will be able to navigate to any folder of interest. **Note:** In Windows 10x I recommend the use of `powershell`. ON a new Anaconda/miniconda install you first open an Anaconda powershell window from the *start* menu, then run `conda init powershell`, to allow any powershell to interact with the `conda` to manage environments and launch jupyter. When this is done rightclick of over the start menu (lower left) and select *Windows PowerShell*.



Once inside the `jupyter lab` interface interface you can easily navigate to a location of interest and then launch an existing notebook or create a new notebook. It is best to start `jupyter lab` at a high level in your directory tree so than you can navigate to where ever you need to.

From the above you can see that the notebook is all set. Note that the first cell is only relevant if you intend to render your notebook to pdf using the LaTeX backend. This requires that you install Pandoc and then an appropriate install of the TeX/LaTeX type setting system. On Windows MikTex or TeXLive, and on macOS and Linus TeXLive. The Pandoc Web Site provides details.

The two import lines just bring in the `scikit-dsp-comm module` `sigsys.py` with alias `ss` into the workspace). By the way, IPython *magics* make general OS path manipulation a breeze. Some of then don't even require that you forst type `%`. You do need to know basic Linux/Unix OS commends. I show you a few examples below:

```
pwd # check your path
```

```
'C:\\Users\\mwickert\\Documents\\Courses\\Tablet\\ece5650'
```

```
# Move up one level
%cd ..
```

```
C:\Users\mwickert\Documents\Courses\Tablet\ece5650
```

```
cd Python_Basics
```

```
C:\Users\mwickert\Documents\Courses\Tablet\ece5650\Python_Basics
```

```
%ls
```

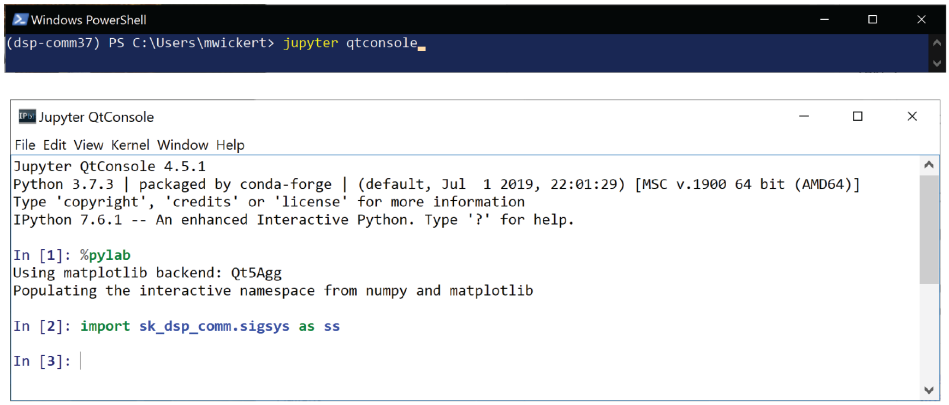
```
 Volume in drive C has no label.
 Volume Serial Number is 003D-B818

 Directory of C:\Users\mwickert\Documents\Courses\Tablet\ece5650\Python_Basics

10/31/2019  10:30 AM    <DIR>          .
10/31/2019  10:30 AM    <DIR>          ..
10/31/2019  10:28 AM    <DIR>          .ipynb_checkpoints
10/31/2019  10:30 AM         2,922,583 Python Basics_2019.ipynb
10/31/2019  10:26 AM    <DIR>          Python_Basics_files
               1 File(s)      2,922,583 bytes
               4 Dir(s)  251,068,715,008 bytes free
```

If you are reading the present document in pdf format, you should consider downloading the notebook version so you can follow along with interactive calculations and experiments, as you learn *Python Basics*.

Finally, the third cell configures the inline graphics display mode. The first line (uncommented at present) uses *scalable vector graphics* (SVG), which produces nice crisp graphics that also render well when the notebook is exported as a markdown (`.md`) file. The second option is often best for direct LaTeX to pdf rendering. This will create *crisp* vector graphics. I recommend using this only when you get ready to export a notebook to pdf. You will have to use `Run All` from the `Cell` menu to convert all graphics to pdf and then switch back later to again have regular inline plots. If you go through the extra trouble of setting up TeX, Pandoc and Inkscape, then you can directly render SVG to LaTeX PDF with one export command.

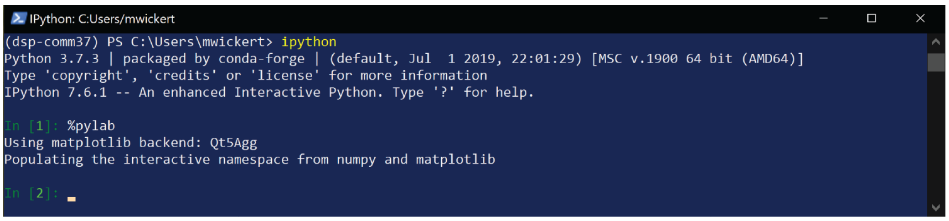
Option 2: Using the QT Console

Starting again from a terminal, this time you launch `jupyter qtconsole` . The result is a very user friendly command line with pop-up help, history scroll-back, and other features.



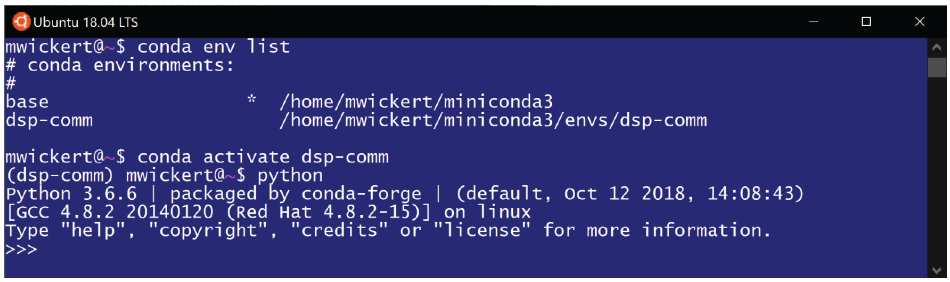
Option 3: Ipython Console

Starting again from a terminal, this time you launch `ipython`. The result is a terminal-based interface to Python with some very nice features, compared to the raw terminal shown next. I frequently run this environment in the termnal window that is embedded in `VS Code`.



Option 4: Raw Python Terminal

Here I am showing the raw Python terminal using Windows Subsystem for Linux (WSL), but it could be in the PowerShell, or the Bash shell of macOS or Linux. This is no frills environment but very good for running Python scripts that perhaps take command line entries.



Finally moving on Python language basics...

Data Types and Simple Calculations

Hello World

The classic first program in any language is *Hello World*. In Python this is accomplished quite simply via:

```
print('Hello Python World!')
```

```
Hello Python World!
```

As I roll through basics be aware that comments begin with `#` and multiline comments look like

```
"""
A multiline comment
The second line
The third line
"""
```

if I need access to a particular Python module I need to import it first:

```
# Here I will import the SciPy signal processing module and
# give it a short name of signal
import scipy.signal as signal
```

More of discussion of `import` and modules will occur later. Until I start talking about numpy I will keep all he topics limited to what you can do with native Python. **Note:** If you need to continue a line you can use `\` (backslash). You cannot break a string this way. You can also break lines at commas.

`float`, `complex`, `long`, `int`, `str`, and `boolean`

The Native Python data types are actually rather few. When Numpy is brought in this changes, but for now that is not the case.

Float

In signals and systems work the `float` type is actually is actually a `double` as found in the C language. This means it consumes 64 bits (8 bytes) of memory. A `float` is created by simply including a decimal point in the number.

```
a = 1.2
b = 4.603
a*b
```

```
5.523599999999999
```

To be sure you can use the built-in function `type()`. To compare several calculation I will string together several calls to `type()` with parenthesis and commas in between. This way I can display the rults all on one line. **Note:** I have actually created a compount type known as a `tuple`. More on *tuples* later.

```
(type(a), type(2), type(2*a)) # note the upcasting to float
```

```
(float, int, float)
```

```
type((type(a), type(2), type(2*a))) # still have to explain tuples
```

```
tuple
```

The native operations available with `float` types are given in the following table. The table order is from lowest to highest precedence.

Native Type Standard Arithmetic Operators

Name	Operator	Quick Example
Addition	+	>>> 3 + 4.5 7.5
Subtraction	-	>>> 10 - 7.3 2.7
Multiply	*	>>> 3.4 * 39.1 132.94
Division	/ Note one number must be a float in Python 2.7	>>> 45.2/89.1 0.5072951739618407
Integer Division	// In Python 3.3 or / In Python 2.7 (see note)	>>> 67//8 or 67/8 8
Remainder	% Note with numpy you typically use mod(a,b)	>>> 13 % 8 5
Exponentiation	**	>>> 3**8 6561

Note: In Python 2.7 Python 3.3 division behavior is available if you make your first import (be careful with this):
`>>> from __future__ import division`

A frustration with the older Python 2.7 is that when you do perform simple division such as

```
x = 6/7
```

thinking you will form a float, you instead get and integer. In the example above you get `0`. In Python 3.3 this is resolved. I am making a big deal about this because over and over again I get tripped up.

So what can you do? In Python 2.7 I find it best to just remember to put a decimal point on one of the two numbers when working with ratios of integers in math calculations. A hard conversion to `float` is possible using the native function `float()`, e.g.,

```
x = 6/7.
# or
x = 6/float(7)
```

The best option now is to use Python 3.6 or or higher

```
(6/7,6//7) # display two results, again using a tuple
```

```
(0.8571428571428571, 0)
```

Note: The above is how Python 3 nehaves and is the new norm when using Python.

Complex

Another standard type to Python is `complex`. For signals and systems work, `complex` plays a significant role. The constant `1j` gives you access to $\sqrt{-1}$, and allows you to readily form complex quantities. In the following example I will again create a `tuple` just for the convenience of displaying multiple results without using a formatted `print` statement.

```
z1 = complex(1.,3) # z = complex(x,y) builds a complex type
z2 = 5. - 20j
(z1 + z2, z1 - z2, z1*z2, z1/z2)
```

```
((6-17j), (-4+23j), (65-5j), (-0.12941176470588237+0.08235294117647059j))
```

The convenience of built-in complex arithmeic is very nice. I need to mention however, that getting access to functions beyond the operators listed in the table above, requires the `import` of specific code *modules*. The `math` and `cmath` bring in additional functions for float or real numbers and complex numbers respectively. Don't get too excited about jumping in to use these modules. With NumPy, which will be talked about later, the use of `math` and `cmath` is taken care of for you. AN with NumPy you will have full vector/matrix support. I just mention it here so you know it does exist, and if for some strange case you don't want to use NumPy, this is what you will have work with.

Int and Long

For integer math, indices, and loop counters, Python has the types `int` and `long`. The `int` type is a *signed* integer similar to `int` in the C language. The size of `int` depends upon the machine you are running on. If you import the `sys` module you can find out more information about `int` for your machine:

```
import sys
sys.maxsize
```

```
9223372036854775807
```

Note: In Python 2.7 you would run `sys.maxint` but in Python 3 this is no longer available.

On a 64-bit OS the maximum value should be like $2^{64-1}-1$, accounting for the fact that one bit is needed for the sign and since zero is represented you have to stop one value short of 2^{63}.

The native math capability of Python goes one step further via the `long` type. The `long` type offers unlimited size! Furthermore if you are working with an `int` type and perform an operation that exceeds the maximum size, it will converted to a `long` integer for you. Loop counters however, are bound to the maximum size on `int`. There are work arounds for this too.

```
x = 34
(type(x),x) # display two results, again using a tuple
```

```
(int, 34)
```

```
y = x**32
(type(y),y) # display two results, again using a tuple
```

```
(int, 1017010285931541177457962846134113802302590130585 6)
```

```
1-y
```

```
-101701028593154117745796284613411380230259013058 55
```

Notice in the above examples that long integers are displayed with `L` appended to the end.

Other Bases

In computer engineering you often need to work with other bases.

Bitwise Operations

Along with the display of integers in other formats, Python also supports the *bitwise* operations shown in the following table.

Native Bitwise Arithmetic Operators

Name	Operator	Quick Example
Bitwise not	~	>>> ~0x001 + 0x100 254 since hex(~0x001) = -0x2
Shift left	<<	>>> bin(0b0001<<2) '0b100' or 4
Shift right	>>	>>> bin(0b11001>>3) '0b11' or 3
Exclusive OR (XOR)	^	bin(0b1010^0b1111) '0b101' or 5
AND	&	>>> bin(0b101100 & 0xf) '0b1100' or 12
OR	|	>>> bin(0b1000 | 0b0001) '0b1001' or 9

Note: If you search the Internet you will find little helper functions to allow you to represent hex values with proper sign extension.

String

String creation and manipulation in Python is a breeze. In signals and systems work string manipulation often shows up when working with formatted printing, on the screen or in a text file, and in labels for plots.

The ability to mix fixed text with formatted numbers, on the fly, is very handy when you have to tabulate analysis and simulation results. Formatted print strings will discussed when I discuss the `print()` function. Presently the basic of type `str` are discussed.

Formally a string in Python is a sequence of *immutable* characters. Immutable means the values of the string cannot be changed. You can easily create a new string from an existing string, and this is where you can introduce changes you may desire.

A string can be indicated using: (1) single quotes (2) double quotes, or (3) triple quotes to create a string block.

```
xa = 'Bat'
xb = "Bat"
xc =\
"""
Many bats flying
through the air.
"""
```

```
xa = 'Bat'
xb = "Bat"
xc =\
"""
Many bats flying
through the air.
"""
# Use a tuple to display some results
(xa,type(xa),xb,xc)
```

```
('Bat', str, 'Bat', '\nMany bats flying \nthrough the air.\n')
```

Note: The multi-line string has embedded line feed, `\n`, characters.

Single and double quotes are interchangeable and are useful when you want to preserve quotes that belong in a string, e.g.,

```
xd = "It's a beautiful day"
xd
```

```
"It's a beautiful day"
```

Don't be afraid to experiment!

String can be *concatenated* or added quite easily:

```
'Hello,' + ' ' + 'what is your name?'
```

```
'Hello, what is your name?'
```

The number of characters in a string can be found using the `len()` function, while individual characters can be accessed using brackets, e.g., `xd[3]`.

Indexing can be used to obtain *substrings*. The indices are integers which run from 0 to `len()-1`. To generate substrings use brackets, i.e.,

```
len(xd)
```

```
20
```

The table below sumarizes basic string manipulation, including the fun topic of slicing. Slicing returns with native Python `lists`, `tuples`, and NumPy `ndarrays`.

Native string operations

Name	Operation	Quick Example
Concatenate/ adding	`xa + xb` to concatenate strings	`>>> 'Hello' + ' ' + 'World'` `'Hello World'`
Replicate/ multiply	`x*n` or `n*x` to replicate a string n times	`>>> 'Bat' * 3` `'BatBatBat'`

Indexing	`x[n]`
	`x[-1]` the end value
	`x[-2]` the second from the end
		`>>> x = 'Bat'`
		`>>> x[1]`
		`'a'`
		`>>> x[-1]`
		`t`

Slicing: Many forms possible	`x[n:m]` the substring from n to `m-1`
	`x[:m]` the substring from `0` to `m-1`
	`x[n:]` the substring from n to the end
	`x[n:-1]` the substring from n to end-1
	`x[n:-2]` the substring from n to end-2
	`x[n:m:k]`
		`>>> x = 'Bright colors'`
		`>>> x[1:6]`
		`'right'`
		`>>> x[7:]`
		`'colors'`
		`>>> x[:-1]`
		`'Bright color'`
		`>>> x[::2]`
		`'Bih oos'`
		In the above the third argument is the optional ***stride*** (the default, if not given is 1) factor which controls the step size as you run from 0 to the end in this case, since only `::` is given.

Note: Indexing and slicing will work the same way when wiring with Python `lists` and `tuples`, and the Numpy `ndarray`.

There are many functions for searching and modifying strings. Too many to cover here. If you feel the need, do some searching on your own. As a specific example, consider breaking a string down into substrings and then put back together in a differnt form. Below I use `find()` to do some simple string parsing to assit in the tear-apart:

```
xd[0:5] + xd[xd.find('beau'):xd.find('day')-1]
```

```
"It's beautiful"
```

Boolean

The `boolean` type has it place in making logical decisions in program flow. In Python the boolean type holds either `True` (1) or `False` (0). You will see `booleans` in action when I discuss program flow. Logical operation as used in program flow control return `booleans`. A few simple examples follow:

```
b1 = True
b1 > 1
```

```
False
```

```
34 > 0 and 4 < 3
```

```
False
```

```
34 > 0 or 4 < 3
```

```
True
```

Data Structures

Python's native data structures of interest here are `lists`, `tuples`, and briefly `dictionaries`. All three of these data structures are *sequences* that can be thought of as containers for Python *objects*. The most import object you will be using is the `ndarray`, which I have made mention of many times. Although note mentioned in the section on string, they are also *sequences* of characters.

Lists

Simply put, a `list` is a *mutable* (changable) sequence of objects. A list is created using brackets with commas to separate the items:

```
l1 = [1,'abc',23.4]
l1
```

```
[1, 'abc', 23.4]
```

Indexing and slicing of lists works the same as with strings. In fact a list can hold strings as you see in the above example. A list can hold a list of lists, so it is very flexible and has a collection of useful functions (actually methods to use object oriented terminology): `append()`, `count()`, `extend()`, `index()`, `insert()`, `pop()`, `remove()`, `reverse()`, and `sort()`. A few examples follow.

Before getting too excited about lists understand that serious number crunching is best done using the `numpy` package. This package brings full vector/matrix manipulation to Python, and is optimated for speed using pre-compiled numerical operators. List operations run at interpreted speed.

```
l1 = [1,'abc',23.4]
l2 = [10.0, 20.0, 50.0]
l1.extend(l2)
l1[:]
```

```
[1, 'abc', 23.4, 10.0, 20.0, 50.0]
```

```
len(l1)
```

```
6
```

```
l1.pop(1)
```

```
'abc'
```

I will introduce `for` loops in detail a little bit later, but I want to show you how you can fill a list with numbers using the list `append()` method and `range()` to set up the iteration.

Once upn a time `range()` was used to create a list of numbers In Python In Python 3 things are a little different. You have to be content with thinking of range as an iterator object of the form:

```
n1 = range(start,stop,step) # = [start,start+step,start+2*step,...]
n2 = range(20) # = [0,1,2,...,19]
```

If `start` is omitted the sequence starts at 0. If `step` is omitted the step size is 1. Note `step` may be negative.

Note: You cannot get yours hands on the individual values `n1` unless you use it with a *loop* of some sort. Note *looping* structures are formally talked about a little later.

Creating a single argument range object results in:

```
n1 = range(10)
n1
```

```
range(0, 10)
```

If first create an empty list, then iterate through the values contained in the range object with a `for` loop, we repeatedly append the list to grow it into a long list. See the below:

```
list1 = []
for k in range(10):
    list1.append(k)
list1
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Now insert a new value at index 4:

```
list1.insert(4,20)
list1
```

```
[0, 1, 2, 3, 20, 4, 5, 6, 7, 8, 9]
```

Lists can contain lists, and so on. Below I create `list2` as a list of two lists made from subsequences of the original `list1`.

```
list2 = [list1[:5],list1[5:]]
list2
```

```
[[0, 1, 2, 3, 20], [4, 5, 6, 7, 8, 9]]
```

Indexing and slicing into `list2` now requires two indices:

```
list2[1][:3]
```

```
[4, 5, 6]
```

List Comprehensions

Indexing through lists and performing calculations is a frequent task, at least without NumPy. Python allows you to combine looping and list manipulation into one operation.

```
new_list = [function_of_index for index in range(n1,n2+1)]
#or to list the values in the terminal immediately
[function_of_index for index in range(n1,n2+1)]
```

Below is a simple example that returns a list of numbers corresponding to $3 + 4n + n^2$ for $0 \leq n \leq 10$.

```
[3+4*n+n**2 for n in range(0,11)]
```

```
[3, 8, 15, 24, 35, 48, 63, 80, 99, 120, 143]
```

When you use list comprehensions you can write very *terse* Python code. I encourage you to explore list comprehensions as you feel more comfortable with the language. With NumPy the list comprehension still provides a convenient way to fill a `list` or `array` with values of interest, but again NumPy has its own ways too, that most likely are even faster.

Tuples

A *tuple* is like a `list`, but it is *immutable* (not changable). Your first reaction to this might be '*what good is it if I can't change it*'. It turns out that the immuatabilty aspect is perfect for the needs of engineering/scientific computing.

Creating a `tuple` can be done using parenthesis much like you do with `lists`. One significant difference is that a single element tuple requires a comma so as not to be confused with the ordinary use of parenthesis.

```
t1 = (1,2,23.5,'abcd')
t1
```

```
(1, 2, 23.5, 'abcd')
```

```
t2 = (27)
t2 # This is not a one element tuple
```

```
27
```

```
t3 = (34.5,) # the comma does it, its a one element tuple
t3
```

```
(34.5,)
```

```
type(t3)
```

```
tuple
```

Trying to change a value of a tuple element fails, as you can see from the following:

```
t1[1] = 56
```

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

<ipython-input-151-4a896955a572> in <module>()
----> 1 t1[1] = 56
```

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

A typical use of the tuple is as a return from a function call. Each element of the tuple can be an object such as a `list` or with NumPy an `ndarray`. You can then *unpack* the tuple into its constituent objects, say a *frequency* array and a *frequency response* array. Further analysis follows.

Suppose you have a function that returns a tuple of two lists. List 1, denoted `l1`, containing numbers and list 2, denoted `l2`, containing characters, you can unpack the tuple into two lists as follows:

```
# First set up the scenario by artificially creating
# a tuple containing two lists
l_composite = ([0,1,2,3,4],['a','b','c','d','e'])
# Break the tuple apart
l1,l2 = l_composite
```

```
l1
```

```
[0, 1, 2, 3, 4]
```

```
l2
```

```
['a', 'b', 'c', 'd', 'e']
```

Tuples have functions, such as `len()`,

Dictionaries

A dictionary is a *mutable* (changable) sequence of `values` that is addressable using a *name/key*. The *key* needs to be unique, but the *value* does not. Dictionaries like lists are mutable.

The motivation for introducing dictionaries at this time is because some of the numerical algorithms in SciPy return dictionaries. If you should need to use one of these algorithms, then you will need to know something about dictionaries.

To create a dictionary using braces to create `{key : value}` pairs.

```
weekdays = {'monday' : 1, 'tuesday' : 2, 'wednesday' : 3,
            'thursday' : 4, 'friday' : 5}
type(weekdays)
```

```
dict
```

You can now access the dictionary elements using the keys:

```
weekdays['wednesday']
```

```
3
```

Dictionaries have a collection of associated functions. For example, you can list the keys using the `keys()` method:

```
weekdays.keys()
```

```
['tuesday', 'thursday', 'friday', 'wednesday', 'monday']
```

If you have a dictionary but don't know whats inside, you can `list()` it as tuples. The order of the list is the *hash* ordering, which is an internal order scheme for fast retrieval.

```
weekdays.items()
```

```
[('tuesday', 2),
 ('thursday', 4),
 ('friday', 5),
 ('wednesday', 3),
 ('monday', 1)]
```

Variables

You have already seen variables in action, but there are some fine points you need to know about them. Variable names can contain characters, numbers, and underscores. Variables cannot begin with a number.

Since everything in Python is an object, all objects have an address. If you declare a structure variable (say a list) it is given an address. If you later set the list variable name equal to the first variable you **will not** be creating a new object. Rather you create a new reference to the same object. Python does have a `copy` method for these instances when you really do want a copy.

```
a1 = [0,23.4]
a2 = a1
(a1,a2)
```

```
([0, 23.4], [0, 23.4])
```

```
a2[0] = 56.8
(a1,a2)
```

```
([56.8, 23.4], [56.8, 23.4])
```

Notice in the above reassignement of the first element of list `a2`, the values held by `a1` have followed. In other words `a1` references the same object. To insure you actually make a copy, you can use some form of copy method. For Numpy `ndarrays` seen later, there is a `copy()` method. For `lists` you can use `a4 = list(a3)` to make a copy:

```
a3 = [2,17]
a4 = list(a3)
(a3,a4)
```

```
([2, 17], [2, 17])
```

```
a4[1] = 20
(a3,a4,'<== It works!')
```

```
([2, 17], [2, 20], '<== It works!')
```

Formatting Strings and Gathering User Input

Strings and gathering user input may seem unrelated, but they come together when you to write interactive programs in Python. I have placed this section here so that it can put to use in the section on flow control coming up next.

Formatting Strings and Printing

Being able to control the format of numers displayed both on the screen and in files allows you to effectively communicate the results of your Python analysis and simulation. Python supports two approaches: *string interpolation* and *formatting strings*. I will be showing just *string interpolation*, as to me it is very easy to pick up, as it follows from a background in C. *Formatting strings* do give more control.

To print a string to the console/terminal, IPython qtconsole, or IPython notebook, you use the `print()` with a formatted string as the argument, or in many cases all rolled up into one statement. A string interpolation expression takes the form `format % values`, where `format` is a string and `values` is a tuple of values. The `%` character present in the string `format` indicates a value should placed in the string using a format specification.

Consider the following simple example:

```
v1 = 3.141516
v2 = 2*v1
print('v1 = %6.4f and v2 = %2.4e' % (v1,v2))
```

```
v1 = 3.1415 and v2 = 6.2830e+00
```

Note: The values to be formatted are contained in the tuple following the `%` character. The formatting for the values always follows a `%` as well. Don't be confused, there are multiple uses of `%` in string interpolation.

If you simply wanted a string for use in plot labels, etc. you can write:

```
str1 = 'v1 = %6.4f and v2 = %2.4e' % (v1,v2)
str1
```

```
'v1 = 3.1415 and v2 = 6.2830e+00'
```

As I said from the start, *string interpolation* is very much like string formatting in C. The format string specifications are given is the table below.

Format specifications for strings and use in `print()`

Character	Type	Example
d	Integer	print('x = %d' % x)
e, E	Engineering notation with e or E respectively for the exponent	print('x = %2.4e' % x)
f	Floating point	print('x = %6.4f' % x)
g	General f or e depending upon need. Decimal point and trailing zeros may be ommitted	print('x = %2.4g' % x)
o	Octal, not too common	print('x = %o' % x)
s	String	print('x = %s' % x)
x, X	Hexadecimal lower or upper case, i.e., 0x or 0X	print('x = %x' % x)

Note: `d`, `f`, and `s` are the most common format types. The `f` specification is very nice for `floats`.

When `for loops` are introduced in the next section you will see how nice tabular lists of data can be prepared. As a quick example which iterates over the list `[-23, 34, 1004]` consider:

```
# Use of format specifications; also not line continue via \
for k in [-23,34,1004]:
    print('Decimal: %o, Decimal padded: %4d, Hex: %x,\
    Hex string: %s' % (k,k,k,hex(k)))
```

```
Decimal: -27, Decimal padded:  -23, Hex: -17,    Hex string: -0x17
Decimal: 42, Decimal padded:   34, Hex: 22,    Hex string: 0x22
Decimal: 1754, Decimal padded: 1004, Hex: 3ec,    Hex string: 0x3ec
```

Gathering User Input

User input may be provided interactively or in the case of a Python script via command line areguments. Both are of interest, with the latter perhaps being more relevant to Python applications running under the control of another program.

Interactive User Input

The function used to accept user inputs is

```
val = input('format string')
```

Below is a simple example:

```
val = input('Enter a number: ')
```

```
Enter a number: 234.5
```

```
(val,type(val))
```

```
(234.5, float)
```

Running Scripts with Command Line Arguments as Inputs

There are times when you may want to write a Python *script* that you can run from the command line or perhaps have another program call. As an example, I have written GUI apps in another language that bring together both command line C++ executables and Python script outputs.

A Python script is a `*.py` file containing code you might ordinarily type at the Python or IPython prompt. You run the script right from the terminal provided by your OS:

```
Marks-MacBook-Pro:IPython_notebooks wickert$ python my_script.py arg1 arg2
arg_etc
```

Note: You can also run scripts from IPython using the `%run` magic, i.e.,

```
In [28]: %run my_script arg1 arg2 arg_etc
```

Note: one or more command line argument may be supplied following the script file name. The script is actually a Python code module that may contain functions as well as a script *body*, that will run from the command line. Any functions in the module can be used by importing the modules *namespace* into your Python (IPython) environment using:

```
import my_script
```

A sample script that reads four command line arguments is given below. This script imports methods from the `sys` module for reading the command line arguments and the `os` module to allow the full path to the script to be discerned. Having the full path comes in handy when you want to read or write files from your script and you have called the script from another directory, say even via another program.

```
#!/usr/bin/python

"""
cmd_line_test.py
A simple command line script program taking four arguments:
string = a file_name, e.g. data_set.txt
   int = an interger loop variable
 float = a calculations variable
 float = a second calculations variable

Note all command line arguments are read as strings, so no
quotes are required.

Mark Wickert, October 2014
"""

# import needed modules and packages
from sys import argv, exit
import os
import numpy as np

#Get the app path for use later
app_path =  os.path.dirname(os.path.realpath(__file__))

"""
Sample command line:
>>>python cmd_line_test.py cmd_test_results.txt 5 109.8 -34.567
"""
# Read command line arguments and convert as needed
if len(argv) < 4+1: # argv[0] is the script name itself
    print('error: Need 4 command line arguments!')
    print('User provided only %d.' % len(argv))
    exit(1)
else:
    out_file = argv[1]
    N_loops = int(argv[2])
    value1 = float(argv[3])
    value2 = float(argv[4])
```

```
# Do something with the collected inputs
print('Echo commandline inputs back to user:')
print('argv[0] = %s' % argv[0])
print('argv[1] = %s' % out_file)
print('argv[2] = %d' % N_loops)
print('argv[3] = %6.4f' % value1)
print('argv[4] = %6.4f' % value2)
# For reading and writing files you may want the full path
print('FYI, the path to your script is:')
print('%s' % app_path)
# Create an empty N_loops x 2 2D array
output_data = np.zeros((N_loops,2))
for k in xrange(N_loops):
    output_data[k,0] = value1 + k*10.0
    output_data[k,1] = (value1 + k*10.0)/value2
np.savetxt(app_path + out_file,output_data)
```

Running the above script from the terminal prompt results in:

```
Marks-MacBook-Pro:IPython_notebooks wickert$ python cmd_line_test.py
sample_output.txt 20 1823.69 -38276.76
Echo commandline inputs back to user:
argv[0] = cmd_line_test.py
argv[1] = sample_output.txt
argv[2] = 20
argv[3] = 1823.6900
argv[4] = -38276.7600
FYI, the path to your script is:
/Users/wickert/Documents/Documents/IPython_notebooks
```

A quick look at the file `sample_output.txt` reveals a nice list of two columns separated by a space.

```
1.82369000000000055e+03 -4.764483723282744027e-02
1.83369000000000055e+03 -4.790609236518451192e-02
1.84369000000000055e+03 -4.816734749754159051e-02
...
```

The complementary Numpy function `loadtxt()` (discussed later) can easily load a text file into ndarrays, using a variety of options.

Note: this script has also used a `numpy` method that makes it easy to write ndarrays to a text file. More will be said about reading and writing ndarrays to files in the NumPy chapter.

Flow Control

The control of program flow is fundamental to moving on just using Python with NumPy. A lot of good analysis can be done without flow control, but sooner or later you need to include some looping and decision logic.

The key operators used for decision logic in Python are shown in the table below.

Logical and boolean comparison operators

Operator	Type	Example
`X < Y`	Less than	5 < 6 returns `True`
`X <= Y`	Less than or equal to	5 <= 4.9 returns `False`
`X > Y`	Greater than	5 > 6 returns `False`
`X >= Y`	Greater than or equal to	5 >= 4.9 returns `True`
`X == Y`	Equal to (same value)	5 == 5.01 returns `False`
`X != Y`	Not equal (also X <> Y in 2.X)	5 != 5.001 returns `True`
`not X`	If X is false then True; else False	`not` (5 < 6) returns `False`
`X or Y`	If X is false then Y; else, X	(5 < 6) `or` (5 != 6) returns `True`
`X and Y`	If X is false then X; else Y	(5 < 4) `and` (5 != 6) returns `False`

This is also where one of the unusual aspects of Python comes to light, that of *code indenting*. Indenting and unindenting code by 4 four spaces is the standard. Python code editors are set up this way, or you can make it so if not.

Indenting must be consistent all the way through a code block inthe IPython notebook or in general in code module file. It is easy to mess up your indenting, so be careful. This is an area that a newcomer is likely to get frustrated with. Hang in there, it gets better with practice.

In this section I cover `if, elif, else` blocks, `for loops`, and `while` loops. What I will leave for self study is `try`, `else`, and `finally` blocks.

If, elif, and else

In Python the core flow control structure is `if`, `elif`, `else`:

```
if condition1:
    block1
elif condition2:
    block2
...
elif conditionN:
    blockN
else:
    elseblock
```

All code blocks must be indented (by convention 4 spaces and not a tab) from the `if`, `elif`, `else` statements. A condition can be passed over by including the `pass` statement in place of an actual block. Coding continues following the `elseblock` by outdenting. No blank lines required. AT first this seems strange, but you get used to it. The *Canopy* code editor as well as the editor used for code in the IPython notebook help get you up to speed.

```
my_value = 10
if my_value <= 4:
    print('I am in the first block!')
elif my_value > 4 and my_value <= 8:
    print('I am in the second block!')
else:
    print('I am in the default block!')
```

```
I am in the default block!
```

```
modeA = 'Green'
modeB = 'hot'
if (modeA.lower() == 'green') and (modeB.lower() != 'cold'):
    print('What I am looking for!')
else:
    print('No match!')
print('Entered a new block due to outdent')
```

```
What I am looking for!
Entered a new block due to outdent
```

For Loops

The `for loop` in Python is differnt from that found in most other languages.

```
for element in sequence:
    ForCodeBlock
# Outside for loop due to outdent. Carry on with the program flow
```

What you see in the above says that a for loop is governed by the `for element in sequence` statement. The words `for` and `in` must appear. How you choose to handle `element` and `sequence` is up to you. The simplest configuration is to let `element = k`, and index variable and define a sequence (`list`) of integers using the Python native `range()` function:

```
for k in range(10)
    print('Index k = %d' % k)
```

As defined above `k` steps over the values in the list, which here has values `0,1,2,...,10-1`. The use of `range()` is convenient since it can generate a sequence of values to iterate over. The `xrange()` function is better still because it does not have to allocate memory for the entire list.

The list you interate over can be most anything. In signals and systems work you typically have a sequence (list) of numbers, integer or floating point. Below I fill a list with floats manually, but once NumPy is on board you will fill `nrrays` by some other means.

Get Two For One by Using the Iterator `enumerate`

When you process float values in a loop your frequently need to use both the sequence `index` and the `value` itself. The loop interation contruct that I really like makes use of the Python iterator `enumerate`. Consider:

```
x = [0, 0.1, 0.2, 0.3, 0.4, 0.5]
for n, xn in enumerate(x):
    print('n = %d and xn = %4.1f' % (n, xn))
```

The iterator `enumerate` returns both an index to `x` and the value at the corresponding index, in that order. Check it out in the notebook:

```
# enumerate() Demo
x = [0, 0.1, 0.2, 0.3, 0.4, 0.5] # An input list of floats
y = [0.0 for i in range(6)] # A list filled using list comprehension
for n, xn in enumerate(x):
    y[n] = xn**2 # note how I make use of both n and xn
    print('n = %d, xn = %4.1f, and y[n] = %4.2f' % (n, xn, y[n]))
```

```
n = 0, xn =  0.0, and y[n] = 0.00
n = 1, xn =  0.1, and y[n] = 0.01
n = 2, xn =  0.2, and y[n] = 0.04
n = 3, xn =  0.3, and y[n] = 0.09
n = 4, xn =  0.4, and y[n] = 0.16
n = 5, xn =  0.5, and y[n] = 0.25
```

Another useful interator is `reversed()`. You can run everything in reverse:

```
# reversed() Demo
x = [0, 0.1, 0.2, 0.3, 0.4, 0.5] # An input list of floats
y = [0.0 for i in range(6)] # A list filled using list comprehension
for n in reversed(range(len(x))):
    y[n] = x[n]**2 # without xn I have to access the list directly
    print('n = %d, xn = %4.1f, and y[n] = %4.2f' % (n, xn, y[n]))
```

```
n = 5, xn =  0.5, and y[n] = 0.25
n = 4, xn =  0.5, and y[n] = 0.16
n = 3, xn =  0.5, and y[n] = 0.09
n = 2, xn =  0.5, and y[n] = 0.04
n = 1, xn =  0.5, and y[n] = 0.01
n = 0, xn =  0.5, and y[n] = 0.00
```

While Loops

The `while` loop is very similar to the `for loop`, but the loop control point is different. Interation is controlled by a logical compare at the top of the loop and exit the loop occurs when the `condition` becomes false. The previous iteration of the loop takes place before you actually jump out of the loop. Also, you must manage the variable used to form the `condition`.

```
while condition:
    while_block
```

While `condition` is `True` looping continues. Infinite looping is also possible

```
# A never ending loop
while True
    while_block
```

As a simple example consider:

```
i = 5 # initialize the looping variable
while i <= 10: # form the True/False condition
    print(i)
    i += 1 # incrementthe looping variable
```

```
5
6
7
8
9
10
```

The Statements `break` and `continue`

When looping using `for` or `while`, you can use `break` to jump out of the loop and continue with normal program flow. You can also use `continue` to skip the remainder of the code in the `for_block` or `while_block` and come around to the next iteration.

```
#break and continue Demo
print('Here I break at 2')
for i in range(4):
    if i == 2: break
    print('i = %d' % i)
print('Here I continue at 2')
for i in range(4):
    if i == 2: continue
    print('i = %d' % i)
```

```
Here I break at 2
i = 0
i = 1
Here I continue at 2
i = 0
i = 1
i = 3
```

Exceptions: `try`, `except`, and `finally` Blocks

Another aspect of flow control is the ability of a program handle runtime errors without *crashing* your program. For the purposes of this intro tutorial. I consider *exception handling* to be a more advanced topic. I am however including some discussion on this topic incase you are looking at code samples you may find on the internet.

In Python exceptional handling is taken care of using `try`, `except`, and `finally` blocks. The idea behind exception handling is to have the program *catch* that an exception has been *raised* or *thrown*, then handle it in a safe way, and finally let the user know something about what happened.

To be completed later...

Functions

To me the heart and soul of any programming language is the ability to write reusable *functions*. In Python functions are written using a `def` construct.

```
def function_name(arguments):    # arguments are optional
    """
    Function docstring to describe the purpose and variable input/output
    """
    function_body                # The function body must be indented
    return one_or_more variables # The use of return is optional
```

Note: The return statement does not have to appear at the end of the function. You can actually return from multiple locations if you need to. The bottom line is the function *does* end when it reaches a `return` statement.

Arguments to the left can be given default values. If say two arguments are given default values and you want to overide the lasgt value only, you must explicity refer to the last value in the function call and give it a value:

```
def my_f1(a,b,c=5,d=25):
    function_body
    return a + b + c + d

# Using the function
x = my_f1(2.3,-4.7)
27.6
y = my_f1(2.3,-4.7,d=20)
22.6
```

```
def my_function(a,b,c=8):
    """
    A simple example function that takes three arguments:
    a = arg1
    b = arg2
    c = arg3, which has a default value of 8
    x = a + 31.5*b/c for c != 0 otherwise
        a + 31.5*b/1000
    """
    # Conditional evaluation
```

```
    if c == 0:
        x = a + 31.25*b/1000
    else:
        x = a + 31.25*b/c
    return x
```

```
# test the function using the default c and with c = 0
(my_function(10.,20.),my_function(10.,20.,0))
```

```
(88.125, 10.625)
```

Object Oriented Python: Writing a Class

Basics

Object oriented programming (OOP) is quite easy in Python. So, what is it and how do you do it? The following subsection walks through a simple examples. Once NumpY is introduced a signal processing I develop the start of a simple filter class.

An object is a collection of data, say scalar numbers, lists, Numpy `ndarrays` , and functions. To create a new object type you first have to create a *class*. The class defines what data types and functions the object will contain. An object is said to *encapsulate* the data and functions that operate on the data.

Objects can *inherent* data and functions from an existing class, if you wish. This can be a very useful property, as it can save you the trouble of starting from scratch if some other class type has much of what you need in your new class.

Writing a Simple Class

In a separate code module or right here in the *Notebook*, you write a class as follows:

```
# Simple starter class entitled Entity for holding name,
# date, and time, and having some methods
# Import some modules needed for your class
import time # a Python standard library module
import datetime # a Python standard library module

class Entity(object): # object is the default to inherit from
    """
    A simple starter class

    Mark Wickert October 2014
    """
    # You begin by initializing the class. This is the class
    # constructor:
    def __init__(self, me): #Note self refers to the object itself
        self.name = me
        # current time since epoch in float seconds
        self.time = time.time()
```

```
        # year, month, date stucture
        self.date = datetime.date.fromtimestamp(self.time)

    # This is a special method that can be implemented to provide
    # a string representation of the object.
    def __str__(self):
        string1 = 'Person %s started at %10.2fs, \n' \
                  % (self.name,self.time)
        string2 = 'which corresponds to year %d, month %d, and day %d.' \
                  % (self.date.year,self.date.month,self.date.day)
        return string1 + string2

    # This is a special method that can be implemented to provide
    # the official representation (repr) of the object.
    # Without it you just get an object address when you type
    # the object name and press enter.
    def __repr__(self):
        return str(self)

    # Create a method to re-set the Entity name.
    def set_name(self,new_name):
        self.name = new_name

    # Time in seconds the Entity has been in service
    def service_time(self):
        return time.time() - self.time
```

Note: You see `self` everywhere when you write a class. All objects and data must be preceeded by `self` and every class method (function) must begin with a reference to `self`. Forgetting `self` somewhere in your class definition is a fairly common error. Be on the look out for this error.

```
# Create a new object of type Entity having name Joe
person1 = Entity('Joe')
```

```
# Use the repr method to give the representation of person1
person1
```

```
Person Joe started at 1414386667.63s,
which corresponds to year 2014, month 10, and day 26.
```

```
# Change the name of person1 using the setter method
person1.set_name('John')
```

```
# Verify that the name change took place
person1
```

```
Person John started at 1414386667.63s,
which corresponds to year 2014, month 10, and day 26.
```

```
str(person1)
```

```
'Person John started at 1414386667.63s, \nwhich corresponds to year 2014, month
10, and day 26.'
```

```
person1.service_time()
```

```
17.271279096603394
```

After Numpy

With Python basics taken care of, now its time to move on to the real focus of using Python for science and engineering. *NumPy* (Numerical Python) is an open-source Python library for numerical computing. When you combine NumPy with *MatPlotLib* and *SciPy*, and the *IPython* console or notebook app, and you really have a very powerful set of tools. [The full NumPy documentation.]

The writing for the NumPy section is far from complete. At present I have placed many tables.

1. Numpy Fundamentals
2. Working with 1D Arrays
   a. Signals
   b. Systems
3. Working with 2D Arrays (Matrices)
4. The signal processing functions of `ssd.py` and `digitalcom.py`
5. A DSP Class using NumPy and Matplotlib

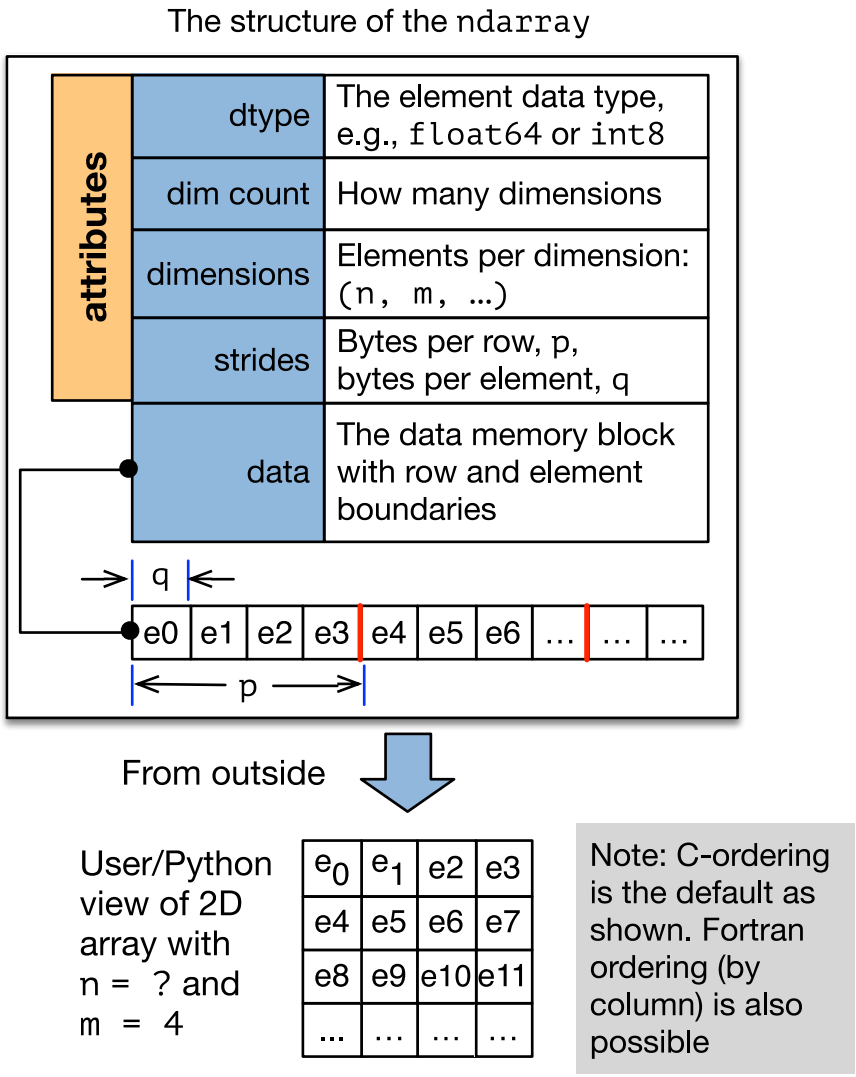
NumPy Fundamentals

The N-Dimensional Array and Available Types

The essence of the ndarray is shown in the figure below. Numerical operations with ndarrays mostly run at full compiled code speed. This is particulary true if the math operations you perform on an array do not change the size of the array.

Once an array is created you can access the attributes and many other methods using the `.` operator. To see the complete list type:

```
dir(numpy)
```



As a quick example consider:

```
# Here I use the `array()` method (see Array Creation below)
A = array([1., 34.,-345, 98.2])
A
```

```
array([   1. ,   34. , -345. ,   98.2])
```

```
A.dtype
```

```
dtype('float64')
```

```
A.shape # This a 1D array
```

```
(4,)
```

When using *PyLab*, which makes the IPython environment similar to MATLAB, you work with ndarrays in a very natural manner. The default data type for floats is double precision or 64 bit (128 bits for complex). Many other data types can be used to make more efficient use of memory. The table below lists the types and makes mention of how you declare types and perform casting from one type to another.

Available `ndarray` data types set by `dtype`

Type	Description
`bool`	Boolean (True or False) stored as a bit
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2**31 to 2**31 - 1)
`int64`	Integer (-2**63 to 2**63 - 1)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 2**32 - 1)
`uint64`	Unsigned integer (0 to 2**64 - 1)
`float16`	Half precision float: sign bit, 5b expo, 10b mantissa
`float32`	Single precision float: sign bit, 8b expo. 23b mantissa
`float64`	Double precision float: sign bit, 11b expo, 52b mantissa
`complex64`	Complex number, represented by two 32-bit floats (real & imag)
`complex128`	Complex number, represented by two 64-bit floats (real & imag)

Note: The highlighted types are the defaults on a 64-bit OS. Type casting is possible using methods such as `y = uint32(x)`, etc.

Array Creation

The table below provides examples of commly used methods to create ndarrays.

Creating NumPy `ndarrays`

Method	Description	Example

`array()`	This is the core method used to create ndarrays from a list. The `dtype` argument is good for setting the per element data type	`>>> a = array([1,2,3,4])`
		will create an int64 array
		`>>> a = array([1,2,3,4],dtype=float16)`
		will create a float16 array

`ones(n1)`
or
`ones((n1,n2))`
	Will create an array of specified length (n1 or n1xn2, etc) containing all ones as 1D,2D, …	>>> a = ones(20)
		a 20 element 1D array
		>>> a = ones((5,4))
		a 5x4 2D array of ones

`zeros()`	Similar to ones() except fills array with zeros	>>> a = zeors(20) a 20 element 1D array
`ones_like()` `zeros_like()`	Create a new array of zeros or ones that replicates the shape of the input argument	>>> a = ones(10) >>> b = zeros_like(a)
Special 1D Array Creation Methods		
`arange([start, ] stop[,step]`	Create an array of values running from `start` to `stop-step`, where `step` is the step size	`>>> x = arange(0,5,0.5)` creates an array of floats [0,0.5,1.0,…,4.5]
`linspace(start,stop,num=50)`	Create an array of linearly spaced of `num` values running from `start` to `stop`	>>> x linspace(1,2,6) creates the array [1.0,1.2,1.4,1.6,1.8,2.0]
`logspace(start, stop, num=50)`	Create an array of log spaced of `num` values running from 10^{start} to 10^{stop}	>>> x = logspace(0,1,10) creates the array [1. , 1.291, 1.668, …, 5.995, 7.743, 10.]

Note: I frequently use `arange()` to create index vectors and initialize arrays using `zeros()` and/or `zeros_like()`.
Tip: If you add `step` to `stop` in `arange()` the final value will be `stop`.

Working With Arrays

Working with arrays is where it's at! You want to solve problems using a technical computing and visualization environment. Working with arrays is how you get your analysis and simulation work done. There are many core functions/methods for this. In the following four tables below I provide some important example methods. Obviously there are many more, and with SciPy and many code modules written by people all over the world, the list goes on and on and on.

A good Web site to go to is PyPI. Not all packages are listed here (mine included at present), but many are. Web searches often end up at this site.

Popular methods for working with `ndarrays`

Function	Description	Example
Logical		
`all()`	True if all elements are nonzero	`>>> x = array([0,1,2,0,4,5])` `all(x) = False`
`any()`	True if any (at least one) elements are nonzero	`>>> x = array([0,1,2,0,4,5])` `any(x) = True`
`find()`	Return the indices where *ravel*(condition) is true	`>>> x = array([0,1,2,2,1,7])` `find(x >= 3) = array([5])`
Slicing 1D arrays (a few cases)		

x[n:m]
x[:m]
x[n:]
x[n:-1]
x[n:-2]
x[n:m:k]
	The 1D subarray from n to m-1
	The 1D subarray from 0 to m-1
	The 1D subarray from n to the end
	The 1D subarray from n to end-1
	The 1D subarray from n to end-2
	The 1D subarray from n to m-1 with k index striding
		`>>> x = array([0,1,2,3,4,5])`
		`x[:2] = array([0,1])`
		`x[::3] = array([0,3])`
		`x[1:-2] = array([1,2])`

Slicing 2D arrays (a few cases)		

x[n:m,j:k]
x[n:m,:]
x[:,j:k]
x[n:m:o,j:k:l]
 	
x[n:-1,:]
x[:,j:-2]
x[3,:]
x[:,0]
	The 2D subarray from n to m-1, j to k-1
	The 2D subarray from 0 to m-1, all columns
	The 2D subarray all rows, columns j to k-1
	The 2D subarray with striding by o and l in rows and columns respectively
	The 2D subarray from n to end-1, all columns
	The 2D subarray all rows, columns j to k-2
	The 2D subarray row 3, all columns
	The 2D subarray all rows, column 0
		`>>> x = array([[0,1,2], [3,4,5]])`
		`x[:2,:2] = array([[0,1],[3,4])`
		`x[-1,-1] = array([[5]])`

Popular methods for working with `ndarrays` (cont.)

Function	Description	Example
Shape & Concatenation		

`reshape()`	Reshape a 1D or 2D array to a new shape; the new shape must be consistent.	`>>> x = arange(0,5)`
		`1D 6 elements`
		`y = reshape(x,(2,3))`
		`2D 2x3 elements`

`concatenate()`	Join a sequence of arrays together. The arrays must have the same shape except in the axis used for combining. axis=0 is rows, axis=1 is columns.	`>>> x = array([[0,1,2,3,4,5]])`
		`2D 1x6 elements`
		`concatenate((x,x), axis=0)`
		`2D 1x6 elements`
		`concatenate((x,x)), axis=1)`
		`2D 1x12 elements`

`hstack()`	Stack arrays horizontally. A subset of `concatenate()`	`>>> x = array([[0,1,2,3,4,5]])`
		`2D 1x6 elements`
		`x = x.T #transpose`
		`y=hstack((x,x))`
		`2D 6x2 columns`

`vstack()`	Stack arrays vertically. A subset of `concatenate()`	`>>> x = array([[0,1,2,3,4,5]])`
		`2D 1x6 elements`
		`y=vstack((x,x))`
		`2D 2x6 columns`

`flatten()`	Values of the argument array become a 1D array. May be done in-place with x.flatten()	`>>> x = array([[0,1,2,3,4,5]])`
		`x.flatten()`
		`1D 6 element`

`transpose()` or `array.T`	Like matrix transpose for 2D arrays. In-place via `x.T`.	`>>> x = array([[0,1,2,3,4,5]])`
		`x.T`
		`2D 6x1 array`

Popular methods for working with `ndarrays` (cont.)

Function	Description	Example
Math	**Many other standard functions, e.g., trig, are also available for array operations**	
`mean(x)`	The sample mean of the values contained in array x.	`>>> x = array([0,1,2,3,4,5])` `mean(x) = 2.5`
`var(x)`	The sample variance of the values contained in array x.	`>>> x = array([0,1,2,3,4,5])` `var(x) = 2.9167`
`std(x)`	The sample standard deviation of the values contained in array x.	`>>> x = array([0,1,2,3,4,5])` `std(x) = 1.7078`
`sum(x)`	The sum of the values contained in array x.	`>>> x = array([0,1,2,3,4,5])` `sum(x) = 15`
`prod(x)`	The sample mean of the values contained in array x.	`>>> x = array([0,1,2,3,4,5])` `prod(x) = 0`
`cumsum(x)`	The sample mean of the values contained in array x.	`>>> x = array([0,1,2,3,4,5])` `cumsum(x) = array([0, 1,3,6,10,15])`
`cumprod(x)`	The sample mean of the values contained in array x.	`>>> x = array([1,1,2,3,4,5])` `cumprod(x) = array([1, 1,2,6,24,120])`
`min(x)`	The sample mean of the values contained in array x.	`>>> x = array([0,1,2,3,4,5])` `min(x) = 0`
`max(x)`	The sample mean of the values contained in array x.	`>>> x = array([0,1,2,3,4,5])` `max(x) = 5`
`conj(x)`	The sample mean of the values contained in array x.	`>>> x = array([2+5j])` `conj(x) = array([2+5j])`

`x.real & x.imag`	The real or imaginary part of the values contained in array x. Also real(x), imag(x)	`>>> x = array([2+5j])`
		`x.real = array([ 2.])`
		`imag(x) = array([ 5.])`

Popular methods for working with ndarrays (cont.)

Function	Description	Example
To and From Files and Conversion		
x.tofile(fname)	Writes an array to a binary file. Assume a float type. This a quick means to save data in binary form, but not very robust.	>>> x = array([0,1,2,3,4,5],dtype-float64) x.tofile('x_arr.bin')
x = fromfile(fname)	Reads an array from a binary file. Assumes a l=float type by default. Undoes the operation of tofile (see above)	>>> x = fromfile(x_arr.bin) returns array([1.,2.,3.,4.,5.])
tolist(x)	Converts an array to a standard Python list. Leave the comforts of ndarrays.	>>> x = array([0,1,2,3,4,5]) x.tolist() returns [0,1,2,3,4,5]
savetxt(x)	Save an array to a text file with rows and columns matching x. Columns space separated.	>>> x = array([[0,1,2,3,4,5],[6,7,8,9,10]) savetxt(x) 2 rows and 6 columns of text
x = loadtxt(fname)	Load a text file into one or more arrays. A very flexible means of reading data sets.	>>> x,y = loadtxt(fname,usecols=(0,2),unpack=True) Take out columns 0 & 2

Graphics and More with Matplotlib

Being able to integrate visualization with engineering calculations is extremely important. In Python this is done using matplotlib. When you import `pylab`, see the first few cells of this document/notebook, matplotlib is brought into your workspace.

Signals and Systems Tools and Examples

• The Scipy Module `scipy.signal`

The full on-line help is here. The function name listing is given below:

```
dir(signal)
Out[31]:
[ 'abcd_normalize', 'absolute_import', 'argrelextrema', 'argrelmax',
 'argrelmin', 'band_dict', 'band_stop_obj', 'barthann', 'bartlett',
 'bench', 'bessel', 'besselap', 'bilinear', 'blackman',
 'blackmanharris', 'bode', 'bohman', 'boxcar', 'bspline', 'bsplines',
 'buttap', 'butter', 'buttord', 'cascade', 'cheb1ap', 'cheb1ord',
 'cheb2ap', 'cheb2ord', 'chebwin', 'cheby1', 'cheby2', 'chirp',
 'cmplx_sort', 'cont2discrete', 'convolve', 'convolve2d',
 'correlate', 'correlate2d', 'cosine', 'cspline1d', 'cspline1d_eval',
 'cspline2d', 'cubic', 'cwt', 'daub', 'decimate', 'deconvolve',
 'detrend', 'dimpulse', 'division', 'dlsim', 'dltisys', 'dstep',
 'ellip', 'ellipap', 'ellipord', 'fftconvolve', 'filter_design',
 'filter_dict', 'filtfilt', 'find_peaks_cwt', 'findfreqs',
 'fir_filter_design', 'firwin', 'firwin2', 'flattop', 'freqresp',
 'freqs', 'freqz', 'gauss_spline', 'gaussian', 'gausspulse',
```

```
 'general_gaussian', 'get_window', 'hamming', 'hann', 'hanning',
 'hilbert', 'hilbert2', 'iirdesign', 'iirfilter', 'impulse',
 'impulse2', 'invres', 'invresz', 'kaiser', 'kaiser_atten',
 'kaiser_beta', 'kaiserord', 'lfilter', 'lfilter_zi', 'lfiltic',
 'lombscargle', 'lp2bp', 'lp2bs', 'lp2hp', 'lp2lp', 'lsim',
 'lsim2', 'lti', 'ltisys', 'medfilt', 'medfilt2d', 'morlet',
 'normalize', 'np', 'nuttall', 'order_filter', 'parzen',
 'periodogram', 'print_function', 'qmf', 'qspline1d',
 'qspline1d_eval', 'qspline2d', 'quadratic', 'remez', 'resample',
 'residue', 'residuez', 'ricker', 's', 'savgol_coeffs',
 'savgol_filter', 'sawtooth', 'scoreatpercentile', 'sepfir2d',
 'signaltools', 'sigtools', 'slepian', 'spectral', 'spline',
 'spline_filter', 'square', 'ss2tf', 'ss2zpk', 'step', 'step2',
 'sweep_poly', 'symiirorder1', 'symiirorder2', 'test', 'tf2ss',
 'tf2zpk', 'triang', 'unique_roots', 'vectorstrength',
 'waveforms', 'wavelets', 'welch', 'wiener', 'windows', 'xrange',
 'zpk2ss', 'zpk2tf']
```

• Using `scikit-dsp-comm`

Folloiw the iunstruction of the `README` at the above scikit-dsp-comm link to clone and install the repository and then begin using it. The support docs for this package are located at read the docs. For the complete index see: index.

In particular the module `sk_dsp_comm.siggsys`, imported at the top of this notebook as

```
import sk_dsp_comm.sigsys as ss
```

was originally written for the book Signals and Systems for Dummies. The contents for this module can be found using `dir(ss)`

```
dir(ssd)
Out[30]:
['BPSK_tx', 'CIC', 'NRZ_bits', 'NRZ_bits2', 'OA_filter',
 'OS_filter', 'PN_gen', 'am_rx', 'am_rx_BPF', 'am_tx',
 'biquad2', 'bit_errors', 'cascade_filters', 'conv_integral',
 'conv_sum', 'cpx_AWGN', 'cruise_control',  'deci24',
 'delta_eps', 'dimpulse', 'downsample', 'drect', 'dstep',
 'env_det', 'ex6_2', 'eye_plot', 'fft', 'fir_iir_notch',
 'from_wav', 'fs_approx', 'fs_coeff', 'ft_approx',
 'interp24', 'line_spectra', 'lms_ic', 'lp_samp',
 'lp_tri', 'm_seq', 'mlab', 'my_psd', 'np', 'peaking',
 'plot_na', 'plt', 'position_CD', 'prin_alias', 'pylab',
 'rc_imp', 'rect', 'rect_conv', 'scatter', 'signal',
 'simpleQuant', 'simple_SA', 'sinusoidAWGN', 'soi_snoi_gen',
 'splane', 'sqrt_rc_imp', 'step', 'ten_band_eq_filt',
 'ten_band_eq_resp', 'to_wav', 'tri', 'unique_cpx_roots',
 'upsample', 'wavfile', 'zplane']
```

• More Modules

The are are many more modules in the `scikit-dsp-comm` package. Visit the `README1` to get the details. There is also the GitHub repo for the Scipy 2017 tutorial that uses this package: SciPy 2017 tutorial.

A Simple DSP Class Case Study

Filters are used frequently in DSP. Filters have characteristics, such as impulse response, frquency response, pole-zero plot. Filters are also used to operate on signals (sequences). You may want to use a filter operate on contiguous blocks/frames of data. When this is done the filter has to maintain state from use-to-use. Lowpass filters are used in decimators and interpolators,

The `class` Code Base

A filter *object* would be nice for keeping all of the above information organized. A preliminary version of the class is implemented below:

```
from __future__ import division #provides float div as x/y and int div as x//y
import numpy as np
import scipy.signal as signal
import sk_dsp_comm as ss
# Create an FIR filter object around the signal.firwin method
class FIR_filter(object):
    """
    An FIR filter class that implements LPF, HPF, BPF, and BSF designs using
    the function signal.firwin.

    Mark Wickert October/November 2014
    """
    def __init__(self,order=20,f_type='lpf',cutoff=(0.1,),fsamp = 1.0,
                 window_type='hamming'):
        """
        Create/instantiate a filter object:
        fir_object = FIR_filter(order,f_type,cutoff=(0.1,),fsamp=1.0,
                     window_type='hamming')

         order = the filter polynomial order; the number of taps is 1 + order
        f_type = the filter type: 'LPF' (lowpass), 'HPF' (highpass),
                 'BPF' (bandpass), or 'BSF' (bandstop)
        cutoff = the cutoff frequency/frequencies in Hz input as a tuple.
                 a pair of cutoff frequencies is needed for BPF and BSF designs
         fsamp = sampling rate in Hz
         window_type = the default is hamming, but others can be found in
                       signal.windows, e.g., hanning (or hann)

        """
        self.N = order # The number of filter taps is N+1
        self.f_type = f_type # 'lpf', 'hpf', 'bpf', 'bsf'
        self.fc = array(cutoff) # The cutoff freq in Hz; two cutoffs for bpf &
bsf
        self.fs = fsamp # In Hz
        # Choose a window from from the type in the signal catalog
        self.window = window_type
        # Design the filter
```

```
        # Note under some circumstances the end coeffients may be almost zero
        # or zero. In these cases trim the filter length and report that the
        # requested filter order was not not achieved. The threshold for
removing
        # coefficients is b_eps
        b_eps = 1e-10
        if f_type.lower() == 'lpf':
            if len(self.fc) == 1:
                self.b = signal.firwin(self.N+1,2*self.fc/self.fs,
                        window=window_type,pass_zero=True)
            else:
                print('For LPF only one cutoff frequency required')
        elif f_type.lower() == 'hpf':
            if len(self.fc) == 1:
                self.b = signal.firwin(self.N+1,2*self.fc/self.fs,
                        window=window_type,pass_zero=False)
            else:
                print('For HPF only one cutoff frequency required')
        elif f_type.lower() == 'bpf':
            if len(self.fc) == 2:
                self.b = signal.firwin(self.N+1,2*self.fc/self.fs,
                        window=window_type,pass_zero=False)
            else:
                print('For BPF two cutoff frequencies required')
        elif f_type.lower() == 'bsf':
            if len(self.fc) == 2:
                self.b = signal.firwin(self.N+1,2*self.fc/self.fs,
                        window=window_type,pass_zero=True)
            else:
                print('For BSF two cutoff frequencies required')
        else:
            print('Filter type must be LPF, HPF, BPF, or BSF')
        #Remove small or zero coefficients from the end of the filter
        if self.b[0] < b_eps and self.b[-1] < b_eps:
            self.b = self.b[1:-1]
            print('Effective/realized filter order = %d' % (len(self.b)-1))
        """
        WRITE ANY ADDITIONAL INITIALIZATION CODE HERE
        """


    def freq_resp(self,mode = 'dB',Npts = 1024):
        """
        A method for displaying the filter frequency response magnitude
        or phase. A plot is produced using matplotlib

        freq_resp(self,mode = 'dB',Npts = 1024)

        mode = display mode: dB magnitude or phase in radians, both versus
               frequency in Hz
        """
        f = np.arange(0,Npts)/(2.0*Npts)
        w,H = signal.freqz(self.b,[1],2*np.pi*f)
        if mode.lower() == 'db':
            plot(f*self.fs,20*np.log10(np.abs(H)))
            xlabel('Frequency (Hz)')
            ylabel('Gain (dB)')
            title('Frequency Response - Magnitude')
        elif mode.lower() == 'linear':
```

```
            """
            Write code here
            """
            pass

        elif mode.lower() == 'phase':
            plot(f,np.angle(H))
            xlabel('Frequency (Hz)')
            ylabel('Phase (rad)')
            title('Frequency Response - Phase')
        elif mode.lower() == 'degrees':
            """
            Write code here
            """
            pass

        elif mode.lower() == 'groupdelay':
            """
            Notes
            -----

            Since this calculation involves finding the derivative of the
            phase response, care must be taken at phase wrapping points
            and when the phase jumps by +/-pi, which occurs when the
            amplitude response changes sign. Since the amplitude response
            is zero the sign changes, the jumps do not alter the group
            delay results.

            Mark Wickert November 2014
            """
            theta = np.unwrap(np.angle(H))
            # Since theta for an FIR filter is likely to have many pi phase
            # jumps too, we unwrap a second time 2*theta and divide by 2
            theta2 = np.unwrap(2*theta)/2.
            theta_dif = np.diff(theta2)
            f_diff = np.diff(f)
            Tg = -np.diff(theta2)/np.diff(w)
            plot(f[:-1],Tg)
            min_Tg = np.min(Tg)
            max_Tg = np.max(Tg)
            ylim([np.floor(np.min([min_Tg,0])),1.2*np.ceil(max_Tg)])
            xlabel('Frequency (Hz)')
            ylabel('Group Delay (samples)')
            title('Frequency Response - Group Delay')
        else:
            print('Error, mode must be "dB" or "phase"')

    def pz_plot(self,auto_scale=True,size=1.5):
        """
        Write doc string
        """
        """
        Write code here
        """
        pass

    def impulse_resp(self):
        """
```

```
        Write doc string
        """
        """
        Write code here
        """
        pass

    def step_resp(self):
        """
        Write doc string
        """
        """
        Write code here
        """
        pass

    def firfilt(self,x,reset=False):
        """
        Write doc string
        """
        """
        Write code here
        """
        pass

    def decimate(self,x,M,reset=False):
        """
        Assuming the filter design is lowpass of the appropriate bandwidth,
        follow LPF filtering with downsampling by M.
        """
        """
        Write code here
        """
        pass

    def interpolate(self,x,L,reset=False):
        """
        Assuming the filter design is lowpass of the appropriate bandwidth,
        upsample by L then LPF filter. A gain scale of L is also included.
        """
        """
        Write code here
        """
        pass
```

The key features of the class at present is that it can design lowpass, highpass, bandpass, and bandstop FIR filters using the *window* method. Once a filter object is created using say

```
fir = FIR_filter(31,'LPF',(100,),1000)
```

you can then use *methods* to plot the frequency response magnitude in dB and the frequency response phase in radians.

Notice that code place holders are present for adding more methods to the class:

1. Not shown impulse response plotting.

2. Not shown step response plotting.
3. Frequency response magnitude linear scale.
4. Frequency response phase in degrees.
5. Pole-zero plot using the function `ssd.zplane`.
6. Filtering of an input sequence $x[n]$ to produce output $y[n]$, with initial conditions maintained should more than one *frame* of data be processed.
7. Decimation of $x[n]$ by the factor M should the filter be an appropriately chosen lowpass filter. The implementation of state maintenance is intended, so again seamless frames processing is possible.
8. Interpolation of $x[n]$ by the factor L should the filter be an appropriately chosen lowpass filter. The implementation of state maintenance is intended, so again seamless frames processing is possible.
9. Not shown is rational number rate changing.
10. Not shown is a means to choose alternate FIR types such as equal-ripple (`remez`) and frequency domain sampling (`fir2`).

Making a Standalone Module

The code has imports listed at the top should you desire to place it in a module by itself. There is one detail missing however. Any of the current commands that plot, i.e., `plot()` or `stem()` will require some rework in a standalone code module. YOu will want to changes the import section of the module to look something like:

```
from matplotlib import pylab
from matplotlib import mlab
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal as signal
import sk_dsp_comm.sigsys as ss
```

All three `matplotlib` imports are needed, but it is `plt` that you will directly work with for doing plotting inside the module. Take a portion of the frequency response plotting method for example. In the following code listing I have added or augmented five lines:

```
def freq_resp(self,mode = 'dB',Npts = 1024):
        f = np.arange(0,Npts)/(2.0*Npts)
        w,H = signal.freqz(self.b,[1],2*np.pi*f)
        plt.figure()  # create a blank figure using the plt object imported
        if mode.lower() == 'db':
            plt.plot(f*self.fs,20*np.log10(np.abs(H)))  #Draw a plot on the plt
object
            plt.xlabel('Frequency (Hz)') #Place a label on the plt object
            plt.ylabel('Gain (dB)')      #Place another label on the plt object
            plt.title('Frequency Response - Magnitude') #Place a title on the
plt object
```

The changes need to be made throught the class definition so it can draw plots when methods are called from `FIR_filter` objects. This of course assumes you have imported the module into your IPython notebook or IPython qt console session.

Lowpass and Bandpass Examples

Try out the class with a few quick examples. I first make a lowpass filter and then a bandpass filter.

```
# Lowpass: N = 31 or 32 Taps, fs = 1000 Hz and fc = 200 Hz
fir1 = FIR_filter(31,'LPF',(200,),1000)
```

```
# Bandpass: N = 64 or 65 Taps, fs = 1000 Hz and fc1 = 200 Hz, fc2 = 300 Hz
fir2 = FIR_filter(64,'BPF',(200,300),1000)
```

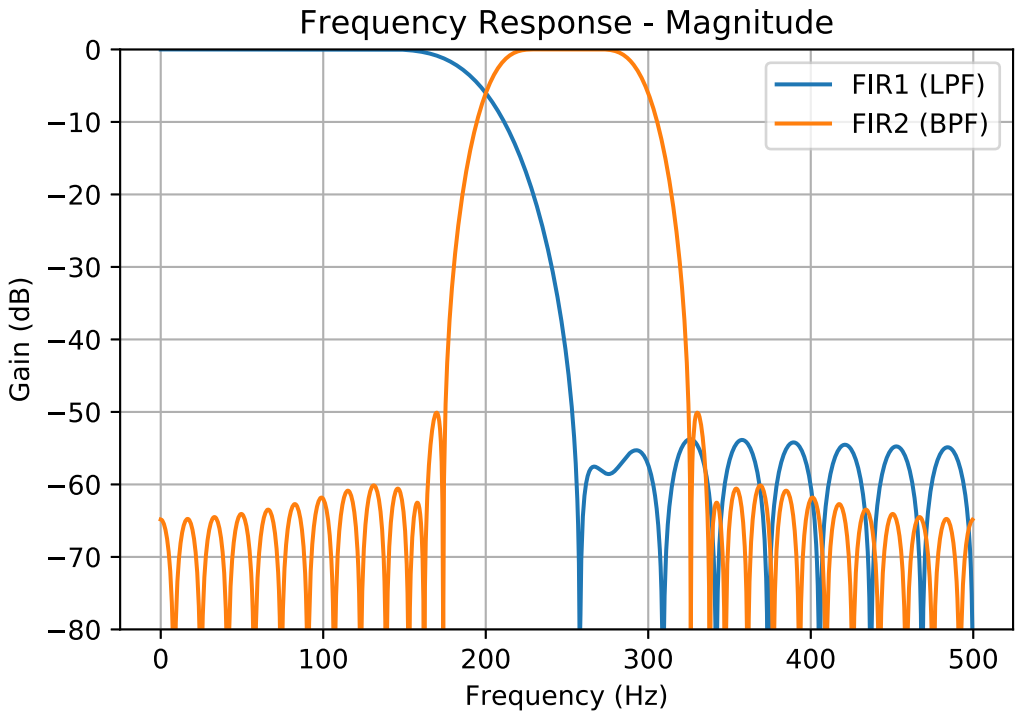
```
Effective/realized filter order = 62
```

You may wonder in the above BPF design what the message *effective filter order of 62 is all about. With the windowed FIR design approach, it is possible for the first and last coefficients to be very small or even zero. This effectively reduces the filter order by two. In the filter constuctor I remove these coefficients to reduce the calculation count and reduce the filter delay.

Frequency Response Magnitude Plots

Verify that the frequency response magnitude in dB method does indeed work.

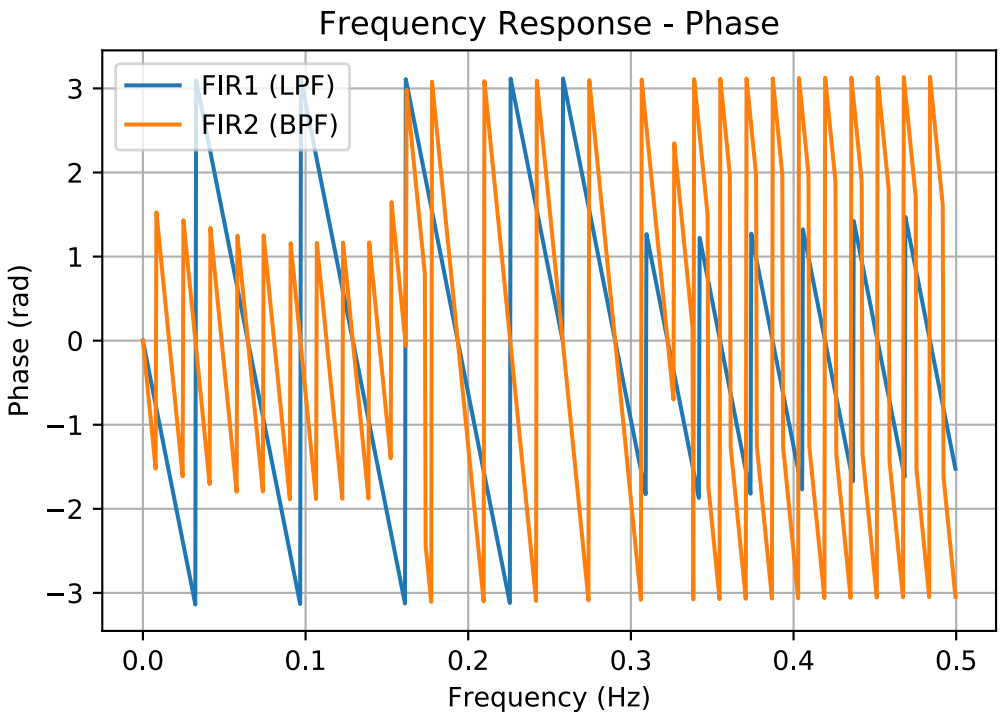
```
fir1.freq_resp()
fir2.freq_resp()
ylim([-80,0])
grid()
legend(((r'FIR1 (LPF)',r'FIR2 (BPF)')),loc='best').get_frame().set_alpha(0.8)
```



Frequency Response Phase Plots

Verify that the frequency response phase in radians method does indeed work.

```
fir1.freq_resp('phase')
fir2.freq_resp('phase')
grid()
legend(((r'FIR1 (LPF)',r'FIR2 (BPF)')),loc='best').get_frame().set_alpha(0.8)
```



Note: The neat matplotlib legend feature (`.get_frame().set_alpha(0.8)`) that allows the transparency so the plot lines can be seen behind the legend frame. Here the opacity is 80% (100% or 1.0) means not opaque.

This is a cross-reference link to Mark Lutz, just to verify that it can be done.

References

Python Converage (core language only no NumPy or SciPy)

[1]: Mark Lutz, *Python Pocket Reference*, 5th edition, O'Reilly, 2014. On Amazon

[2]: Toby Donaldson, *Python: Visual QuickStart Guide*, Third Edition, Peachpit Press, 2014. On Amazon

NumPy/SciPy Python Converage

[3]: Shai Vaingast, *Beginning Python Visualization Crafting Visual Transformation Scripts*, 2nd edition, Apress, 2014. On Amazon

[4]: Python Scientific Lecture Notes. I suggest you download the one page per side pdf version.

[5]: Hans Petter Langtangen, *A Primer on Scientific Programming with Python*, 3rd edition, Springer, 2012. On Amazon

[6]: Ivan Idris, *NumPy Beginner's Guide* 2nd Edition, PACKT PUBLISHING, 2013. On Amazon