

# brief\_tutorial

September 16, 2014

This [IPython](#) notebook is a direct translation of Chapter 1 of

D. J. Higham and N.J. Higham.  
[MATLAB Guide](#), Second edition,  
Society for Industrial and Applied Mathematics, Philadelphia, PA, USA,  
2005, ISBN 0-89871-578-4

with MATLAB code converted to equivalent Python/Numpy (and IPython/Matplotlib/Scipy) code. There are a small number of additions in the text to address Python/Numpy/IPython differences and additions. You can find out more about IPython and the IPython Notebook [here](#) and in reference [2]. All the errors here are, of course, mine. I hope you find it useful.

Don MacMillen: don (sometimes) at macmillen dot net

## 1 Chapter 1 A Brief Tutorial

The best way to learn Python/Numpy is by trying it yourself, and hence we begin with a whirlwind tour. Working through the examples below will give you a feel for the way that Python/Numpy operates and an appreciation of its power and [\[U+FB02\]](#)exibility.

The tutorial is entirely independent of the rest of the book—all the Python/Numpy features introduced are discussed in greater detail in the subsequent chapters. Indeed, in order to keep this chapter brief, we have not explained all the functions used here. You can use the index in [\[1\]](#) to [\[U+FB01\]](#)nd out more about particular topics that interest you.

We will be using the IPython interactive shell as well as the IPython notebook (which is what you are reading now) throughout this tutorial. You can install these by following the directions at [IPython install](#). This tutorial contains commands for you to type at the IPython command line. Alternatively, if you are viewing this notebook in a “live” mode served from a local IPython notebook server, you can directly modify any of the cells and rerun the cell. Just be aware that in some cases you may need to choose the “Run All” option from the cell drop down menu to satisfy intercell dependences.

In the last part of this brief tutorial we give examples of script files and functions. These [\[U+FB01\]](#)les are short, so you can type them in quickly or cut and paste them into an IPython shell (using the `%cpaste` magic), or you can just modify values in place in the IPython notebook and re-evaluate the cell. You should experiment as you proceed, keeping the following points in mind.

- Upper and lower case characters are not equivalent (Python/Numpy are case sensitive).
- Typing the name of a variable will cause IPython to display its current value.
- Python/Numpy uses parentheses, ( ), square brackets, [ ], and curly braces, { }, and these are not interchangeable.
- In the IPython shell, the up arrow and down arrow keys can be used to scroll through your previous commands. Also, an old command can be recalled by typing the first few characters followed by up arrow. You can also use ctrl-p (type p while holding the control key down) to scroll back through the command history stack.
- You can type `help(topic)` to access online help on the command, function, or object topic. Note that hyperlinks, indicated by underlines, are provided that will take you to related help items and the Help browser.

- If you press the tab key after partially typing a function or variable name, IPython will attempt to complete it, offering you a selection of choices if there is more than one possible completion.
- You can quit IPython by typing exit or quit or ctrl-d twice.

Having entered the IPython command, you should work through this tutorial by typing in the text that appears after the IPython prompt, 'In [n]:' where n is a number which indicates the command's location in the command stack, in the command window. After showing you what to type, we display the output that is produced. We begin with arrays. In native Python arrays are lists denoted with the square bracket syntax. However, we want Numpy arrays, so first you import the numpy module and shorten the prefix to "np" (this is a common usage convention)

```
In [2]: %matplotlib inline
import numpy as np
a = np.array([1, 2, 3])
a
```

```
Out[2]: array([1, 2, 3])
```

If you type in the three lines listed in the box 'In[1:]' above into the IPython shell, then you will see the result listed in 'Out[1:]'

This example sets up a 3 element array, which is also called a vector. In some other languages, a distinction is made between a row vector and a column vector, but that is not the case in Numpy. If you define a new vector c

```
In [3]: c = np.array([4, 5, 6])
c
```

```
Out[3]: array([4, 5, 6])
```

Now you can multiply the arrays a and c:

```
In [4]: a*c
```

```
Out[4]: array([ 4, 10, 18])
```

Here, you performed an element by element mulitpy of the two vectors, not the scalar or dot product that some other languages would have done. Notice that in this example, the product a\*c was not explicitly assigned to any variable. When there is no explicit assignment, IPython automatically assigns the result of the expression to the variable named '\_' (underscore) as you can see by the following:

```
In [5]: _
```

```
Out[5]: array([ 4, 10, 18])
```

If you wanted to get the dot product of the two vectors, use the dot function from numpy, as in the following

```
In [6]: np.dot(a, c)
```

```
Out[6]: 32
```

Inputs to Python functions are speci[U+FB01]ed after the function name and within parentheses, as you just saw above with the 'dot' function from the numpy module. You may also form the outer product of the two vectors by calling another function from numpy

```
In [7]: A = np.outer(c, a)
A
```

```
Out[7]: array([[ 4,  8, 12],
               [ 5, 10, 15],
               [ 6, 12, 18]])
```

Here the answer is a 3-by-3 array that has been assigned to A.

The product `a * a`, since the `*` operation is element-wise, is equivalent to squaring a, which uses the `**` operator

```
In [8]: a * a
```

```
Out[8]: array([1, 4, 9])
```

```
In [9]: b = a ** 2
        b
```

```
Out[9]: array([1, 4, 9])
```

Arithmetic operations on matrices and vectors come in two distinct forms. Array sense operations are designed to act elementwise and, as mentioned before, are the default behavior of Numpy.

Matrix sense operations are based on the normal rules of linear algebra and are obtained with either matrix objects, or calling the specific matrix function from numpy. For matrix objects, the usual symbols `+`, `-`, `*`.

```
In [10]: am = np.matrix(a)
         cm = np.matrix(c)
         am * cm.T
```

```
Out[10]: matrix([[32]])
```

Notice that you had to take the transpose of `c` in order to make it into a column vector. For numpy matrix objects, there is a difference between row and column vectors. Also, matrix object can only have one or two dimensions. For instance, the numpy function `ones` takes a tuple and returns a numpy array of that dimensionality. The following will give us a 3 by 3 by 3 array of ones.

```
In [11]: g = np.ones( (3, 3, 3) )
        g
```

```
Out[11]: array([[[ 1.,  1.,  1.],
                 [ 1.,  1.,  1.],
                 [ 1.,  1.,  1.]],
                [[ 1.,  1.,  1.],
                 [ 1.,  1.,  1.],
                 [ 1.,  1.,  1.]],
                [[ 1.,  1.,  1.],
                 [ 1.,  1.,  1.],
                 [ 1.,  1.,  1.]])
```

If you try to turn this into a matrix object, you will get an error

```
In [12]: #h = np.matrix(g)
         print "Error has a problemjQuery203048471758025698364_1409730694459?"
```

```
Error has a problemjQuery203048471758025698364_1409730694459?
```

Here you see how errors are displayed in Python. You see the call stack from the top to the bottom and the error message that is finally displayed.

Numpy has many mathematical functions that operate on arrays element wise when given an array argument. For example,

```
In [13]: np.exp(a)
Out[13]: array([ 2.71828183,  7.3890561 , 20.08553692])
In [14]: np.log(_)
Out[14]: array([ 1.,  2.,  3.])
In [15]: np.sqrt(a)
Out[15]: array([ 1.          ,  1.41421356,  1.73205081])
```

By default, Numpy displays floating point numbers to 8 decimal digits, but always stores numbers and computes to the equivalent of 16 decimal digits. The output format can be changed using the `set_printoptions` function

```
In [16]: np.set_printoptions(precision=4)
         print (np.sqrt(a))

[ 1.          1.4142  1.7321]
```

You can find out much more about the function `set_printoptions` by using IPython's help command

```
In [17]: help(np.set_printoptions)
```

Help on function `set_printoptions` in module `numpy.core.arrayprint`:

```
set_printoptions(precision=None, threshold=None, edgeitems=None, linewidth=None, suppress=None, nanstr='nan',
                 infstr='inf')
    Set printing options.
```

These options determine the way floating point numbers, arrays and other NumPy objects are displayed.

Parameters  
-----

`precision` : int, optional  
 Number of digits of precision for floating point output (default 8).  
`threshold` : int, optional  
 Total number of array elements which trigger summarization rather than full repr (default 1000).  
`edgeitems` : int, optional  
 Number of array items in summary at beginning and end of each dimension (default 3).  
`linewidth` : int, optional  
 The number of characters per line for the purpose of inserting line breaks (default 75).  
`suppress` : bool, optional  
 Whether or not suppress printing of small floating point values using scientific notation (default False).  
`nanstr` : str, optional  
 String representation of floating point not-a-number (default nan).  
`infstr` : str, optional

String representation of floating point infinity (default inf).  
formatter : dict of callables, optional

If not None, the keys should indicate the type(s) that the respective formatting function applies to. Callables should return a string. Types that are not specified (by their corresponding keys) are handled by the default formatters. Individual types for which a formatter can be set are::

- 'bool'
- 'int'
- 'timedelta' : a 'numpy.timedelta64'
- 'datetime' : a 'numpy.datetime64'
- 'float'
- 'longfloat' : 128-bit floats
- 'complexfloat'
- 'longcomplexfloat' : composed of two 128-bit floats
- 'numpy\_str' : types 'numpy.string\_' and 'numpy.unicode\_'
- 'str' : all other strings

Other keys that can be used to set a group of types at once are::

- 'all' : sets all types
- 'int\_kind' : sets 'int'
- 'float\_kind' : sets 'float' and 'longfloat'
- 'complex\_kind' : sets 'complexfloat' and 'longcomplexfloat'
- 'str\_kind' : sets 'str' and 'numpystr'

See Also

-----

get\_printoptions, set\_string\_function, array2string

Notes

-----

'formatter' is always reset with a call to 'set\_printoptions'.

Examples

-----

Floating point precision can be set:

```
>>> np.set_printoptions(precision=4)
>>> print np.array([1.123456789])
[ 1.1235]
```

Long arrays can be summarised:

```
>>> np.set_printoptions(threshold=5)
>>> print np.arange(10)
[0 1 2 ..., 7 8 9]
```

Small results can be suppressed:

```
>>> eps = np.finfo(float).eps
>>> x = np.arange(4.)
>>> x**2 - (x + eps)**2
```

```

array([-4.9304e-32, -4.4409e-16,  0.0000e+00,  0.0000e+00])
>>> np.set_printoptions(suppress=True)
>>> x**2 - (x + eps)**2
array([-0., -0.,  0.,  0.])

```

A custom formatter can be used to display array elements as desired:

```

>>> np.set_printoptions(formatter={'all':lambda x: 'int: '+str(-x)})
>>> x = np.arange(3)
>>> x
array([int: 0, int: -1, int: -2])
>>> np.set_printoptions() # formatter gets reset
>>> x
array([0, 1, 2])

```

To put back the default options, you can use:

```

>>> np.set_printoptions(edgeitems=3,infstr='inf',
... linewidth=75, nanstr='nan', precision=8,
... suppress=False, threshold=1000, formatter=None)

```

You set precision back to 8 to get the default behavior

```

In [18]: np.set_printoptions(precision=8)
         np.sqrt(a)

Out[18]: array([ 1.          ,  1.41421356,  1.73205081])

```

Large or small numbers are displayed in exponential notation, with a power of 10 scale factor preceded by e:

```

In [19]: 2 ** -24

Out[19]: 5.960464477539063e-08

```

Various data analysis functions are also available.

```

In [20]: b.sum()

Out[20]: 14

In [21]: c.mean()

Out[21]: 5.0

```

Here you see an example of the object oriented nature of Numpy. All Numpy arrays are of type “ndarray” and have many methods associated with them. You can list them all by using a ‘dir’ command in IPython, use the following

```

In [22]: dir(a)

Out[22]: ['T',
         '__abs__',
         '__add__',
         '__and__',
         '__array__',
         '__array_finalize__',

```

```
'__array_interface__',
'__array_prepare__',
'__array_priority__',
'__array_struct__',
'__array_wrap__',
'__class__',
'__contains__',
'__copy__',
'__deepcopy__',
'__delattr__',
'__delitem__',
'__delslice__',
'__div__',
'__divmod__',
'__doc__',
'__eq__',
'__float__',
'__floordiv__',
'__format__',
'__ge__',
'__getattr__',
'__getitem__',
'__getslice__',
'__gt__',
'__hash__',
'__hex__',
'__iadd__',
'__iand__',
'__idiv__',
'__ifloordiv__',
'__ilshift__',
'__imod__',
'__imul__',
'__index__',
'__init__',
'__int__',
'__invert__',
'__ior__',
'__ipow__',
'__irshift__',
'__isub__',
'__iter__',
'__itruediv__',
'__ixor__',
'__le__',
'__len__',
'__long__',
'__lshift__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__neg__',
'__new__',
```

'\_\_nonzero\_\_',  
'\_\_oct\_\_',  
'\_\_or\_\_',  
'\_\_pos\_\_',  
'\_\_pow\_\_',  
'\_\_radd\_\_',  
'\_\_rand\_\_',  
'\_\_rdiv\_\_',  
'\_\_rdivmod\_\_',  
'\_\_reduce\_\_',  
'\_\_reduce\_ex\_\_',  
'\_\_repr\_\_',  
'\_\_rfloordiv\_\_',  
'\_\_rlshift\_\_',  
'\_\_rmod\_\_',  
'\_\_rmul\_\_',  
'\_\_ror\_\_',  
'\_\_rpow\_\_',  
'\_\_rrshift\_\_',  
'\_\_rshift\_\_',  
'\_\_rsub\_\_',  
'\_\_rtruediv\_\_',  
'\_\_rxor\_\_',  
'\_\_setattr\_\_',  
'\_\_setitem\_\_',  
'\_\_setslice\_\_',  
'\_\_setstate\_\_',  
'\_\_sizeof\_\_',  
'\_\_str\_\_',  
'\_\_sub\_\_',  
'\_\_subclasshook\_\_',  
'\_\_truediv\_\_',  
'\_\_xor\_\_',  
'all',  
'any',  
'argmax',  
'argmin',  
'argpartition',  
'argsort',  
'astype',  
'base',  
'byteswap',  
'choose',  
'clip',  
'compress',  
'conj',  
'conjugate',  
'copy',  
'ctypes',  
'cumprod',  
'cumsum',  
'data',  
'diagonal',  
'dot',



```
'dtype',
'dump',
'dumps',
'fill',
'flags',
'flat',
'flatten',
'getfield',
'imag',
'item',
'itemset',
'itemsizes',
'max',
'mean',
'min',
'nbytes',
'ndim',
'newbyteorder',
'nonzero',
'partition',
'prod',
'ptp',
'put',
'ravel',
'real',
'repeat',
'reshape',
'resize',
'round',
'searchsorted',
'setfield',
'setflags',
'shape',
'size',
'sort',
'squeeze',
'std',
'strides',
'sum',
'swapaxes',
'take',
'tofile',
'tolist',
'tostring',
'trace',
'transpose',
'var',
'view']
```

To find out more about any of these methods, use the help command

```
In [23]: help(a.argmax)
```

Help on built-in function `argmax`:

```
argmax(...)
    a.argmax(axis=None, out=None)

Return indices of the maximum values along the given axis.

Refer to 'numpy.argmax' for full documentation.

See Also
-----
numpy.argmax : equivalent function
```

```
In [24]: np.pi
```

```
Out[24]: 3.141592653589793
```

```
In [25]: _
```

```
Out[25]: 3.141592653589793
```

```
In [26]: y = np.tan(np.pi/6)
          y
```

```
Out[26]: 0.57735026918962573
```

The Variable `np.pi` is a permanent Variable with value  $\pi$ . The variable “\_” always contains the most recent unassigned expression, as described earlier, so after the assignment to `y`, “\_” still holds the Value  $\pi$ .

You may set up a two dimensional array by concatenating columns using the `np.c_[ ]` array notation

```
In [27]: B = np.c_[[-3, 2, -1], [0, 5, 4], [-1, -7, 8]]
          B
```

```
Out[27]: array([[ -3,  0, -1],
                [ 2,  5, -7],
                [-1,  4,  8]])
```

At the heart of Numpy is a powerful range of linear algebra functions. For example, recalling that `c` is a 3-by-1 Vector, you may wish to solve the linear system  $B * x = c$ . This can be done with the `solve` function from the `numpy.linalg` module

```
In [28]: import numpy.linalg as nl
          x = nl.solve(B, c)
          x
```

```
Out[28]: array([-1.29953917,  1.37788018, -0.10138249])
```

```
In [29]: nl.norm(np.dot(B,x) - c) / (nl.norm(B) * nl.norm(x))
```

```
Out[29]: 3.6020385703775424e-17
```

Some times we see a 0 here, but we usually expect to see something nonzero because of rounding errors. The eigenvalues of `B` can be found using `eig` from the `numpy.linalg` module

```
In [30]: np.set_printoptions(precision=5) # make it the m*lab default
          w, _ = nl.eig(B)
          w
```

```
Out[30]: array([-3.13605+0.j          ,  6.56803+5.10454j,  6.56803-5.10454j])
```

Notice two things about this code snippet. First, `nl.eig` returns two values, `w` are the eigenvalues and `v` would be the eigenvectors. Since in this instance we did not want the eigenvectors we set it to the underscore `'_'`, which is a common idiom in Python. Finally, the solution shows the `j` is the imaginary unit,  $\sqrt{-1}$ . To get the eigenvectors you can do the calculation again

```
In [31]: w, v = nl.eig(B)
         v
```

```
Out[31]: array([[ -0.98290+0.j      ,  0.03854+0.03928j,  0.03854-0.03928j],
               [ 0.12656+0.j      ,  0.80053+0.j      ,  0.80053-0.j      ],
               [-0.13372+0.j      , -0.16831-0.57254j, -0.16831+0.57254j]])
```

The columns of `v` are the eigenvectors of `B`.

To get vectors of evenly spaced values, use the `np.arange` function

```
In [32]: v = np.arange(6)
         v
```

```
Out[32]: array([0, 1, 2, 3, 4, 5])
```

Note that the default starting value is zero. Also, as noted earlier, all Numpy arrays start at index zero and end at index `n - 1` for an array of size `n`.

Nonunit increments can be specified by a third number, often called the 'stride'.

```
In [33]: w = np.arange(2, 10, 3)
         w
```

```
Out[33]: array([2, 5, 8])
```

```
In [34]: y = np.arange(1, 0, -0.25)
         y
```

```
Out[34]: array([ 1.   ,  0.75,  0.5   ,  0.25])
```

This last example illustrates a peculiarity of Python and numpy, that the ending value or index is actually one position short of what you might expect. In the previous example, to have the array actually increment down to zero, you have to specify an end value that is strictly *less* than 0 (but greater than  $-(0.25 + \text{delta})$ ). Here we just use `-0.001`.

```
In [35]: y = np.arange(1, -0.001, -0.25)
         y
```

```
Out[35]: array([ 1.   ,  0.75,  0.5   ,  0.25,  0.   ])
```

You may construct big matrices out of little ones by using the numpy functions `hstack` and `vstack` and the previously mentioned `c_[]` and `r_[]` notations

```
In [36]: C = np.c_[A, [8, 9, 10]]
         C
```

```
Out[36]: array([[ 4,  8, 12,  8],
               [ 5, 10, 15,  9],
               [ 6, 12, 18, 10]])
```

```
In [37]: D = np.r_[B, B, B]
         D
```

```
Out[37]: array([[ -3,  0, -1],
               [  2,  5, -7],
               [-1,  4,  8],
               [-3,  0, -1],
               [  2,  5, -7],
               [-1,  4,  8],
               [-3,  0, -1],
               [  2,  5, -7],
               [-1,  4,  8]])
```

```
In [38]: E = np.vstack((B, B, a))
         E
```

```
Out[38]: array([[ -3,  0, -1],
               [  2,  5, -7],
               [-1,  4,  8],
               [-3,  0, -1],
               [  2,  5, -7],
               [-1,  4,  8],
               [  1,  2,  3]])
```

Notice well that `np.vstack` takes a *single* argument that is a tuple of arrays. That way you can stack as many arrays as needed with a single call to `vstack` or just use the `r_[]` notation.

The element in row `i` and column `j` of the matrix `C` (where `i` and `j` always start at 0) can be accessed as `c[i, j]`

```
In [39]: C[1, 2]
```

```
Out[39]: 15
```

More generically, `C[i1:i2, j1:j2]` picks out the submatrix formed by the intersection of rows `i1` to `i2-1` and columns `j1` to `j2-1`. This type of operation on arrays is generally called *slicing*.

```
In [40]: C[1:3, 0:2]
```

```
Out[40]: array([[ 5, 10],
               [ 6, 12]])
```

You can build certain types of matrices automatically. For example, identities and matrices of zeros and ones can be constructed with `eye`, `zeros`, and `ones`:

```
In [41]: I3 = np.eye(3)
         I3
```

```
Out[41]: array([[ 1.,  0.,  0.],
               [ 0.,  1.,  0.],
               [ 0.,  0.,  1.]])
```

```
In [42]: Y = np.zeros((3,5))
         Y
```

```
Out[42]: array([[ 0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.]])
```

```
In [43]: Z = np.ones((2,2))
         Z
```

```
Out[43]: array([[ 1.,  1.],
               [ 1.,  1.]])
```

Note that these functions take a single argument which is a tuple (except for `np.eye`, which has a different interface). The first argument of the tuple specifies the size of the first dimension of the array and so on.

The methods `rand` and `randn` on an object returned from `numpy.random.RandomState()` do not take tuples. They are convenience functions that have a similar interface to another math environment. These two methods generate random entries from the uniform distribution over  $[0,1]$  and the normal  $(0,1)$  (ie zero mean with unit variance) distribution, respectively.

Note that it is always good for you to initialize a new instance of `RandomState` when using random numbers. This ensures that the random stream is for you alone, and you can reset the seed to make the sequence repeatable. Here we set the starting seed to 20.

```
In [44]: rn = np.random.RandomState() # initialize a new RandomState object
         rn.seed(20)
         F = rn.rand(3, 3)
         F
```

```
Out[44]: array([[ 0.58813,  0.89771,  0.89153],
               [ 0.81584,  0.03589,  0.69176],
               [ 0.37868,  0.51851,  0.65795]])
```

```
In [45]: G = rn.randn(1, 5)
         G
```

```
Out[45]: array([[-0.62064, -0.83453,  0.91636,  0.70784,  0.41968]])
```

At this point several variables have been created in the workspace. You can obtain a list with the `%who` command, where the `%` indicates an IPython ‘magic’ command.

```
In [46]: %who
```

A	B	C	D	E	F	G	I3	Y
Z	a	am	b	c	cm	g	nl	np
rn	v	w	x	y				

The `%whos` magic will additionally give the types and some additional information

```
In [47]: %whos
```

Variable	Type	Data/Info
A	ndarray	3Lx3L: 9 elems, type ‘int32’, 36 bytes
B	ndarray	3Lx3L: 9 elems, type ‘int32’, 36 bytes
C	ndarray	3Lx4L: 12 elems, type ‘int32’, 48 bytes
D	ndarray	9Lx3L: 27 elems, type ‘int32’, 108 bytes
E	ndarray	7Lx3L: 21 elems, type ‘int32’, 84 bytes
F	ndarray	3Lx3L: 9 elems, type ‘float64’, 72 bytes
G	ndarray	1Lx5L: 5 elems, type ‘float64’, 40 bytes
I3	ndarray	3Lx3L: 9 elems, type ‘float64’, 72 bytes
Y	ndarray	3Lx5L: 15 elems, type ‘float64’, 120 bytes
Z	ndarray	2Lx2L: 4 elems, type ‘float64’, 32 bytes
a	ndarray	3L: 3 elems, type ‘int32’, 12 bytes
am	matrix	[[1 2 3]]
b	ndarray	3L: 3 elems, type ‘int32’, 12 bytes

```

c      ndarray      3L: 3 elems, type 'int32', 12 bytes
cm     matrix       [[4 5 6]]
g      ndarray      3Lx3Lx3L: 27 elems, type 'float64', 216 bytes
nl     module       <module 'numpy.linalg' fr<...>mpy\linalg\__init__.pyc'>
np     module       <module 'numpy' from 'C:\<...>ages\numpy\__init__.pyc'>
rn     RandomState  <mtrand.RandomState object at 0x0000000003563978>
v      ndarray      6L: 6 elems, type 'int32', 24 bytes
w      ndarray      3L: 3 elems, type 'int32', 12 bytes
x      ndarray      3L: 3 elems, type 'float64', 24 bytes
y      ndarray      5L: 5 elems, type 'float64', 40 bytes

```

Like most languages Python has loop constructs. The following example uses a for loop to evaluate the continued fraction

$$\begin{array}{c}
 1 \\
 \hline
 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}}}}}}}
 \end{array}$$

which approximates the golden ratio,  $(1 + \sqrt{5})/2$ . The evaluation is done from the bottom up:

```

In [48]: g = 2.
         for k in range(10):
             g = 1 + 1 / g
         g

```

```
Out[48]: 1.6180555555555556
```

Many constants can be found in the scipy submodule constants. You can access them in the following way

```

In [49]: import scipy.constants as sconst
         sconst.golden

```

```
Out[49]: 1.618033988749895
```

Loops involving the while statement can be found later in this tutorial.

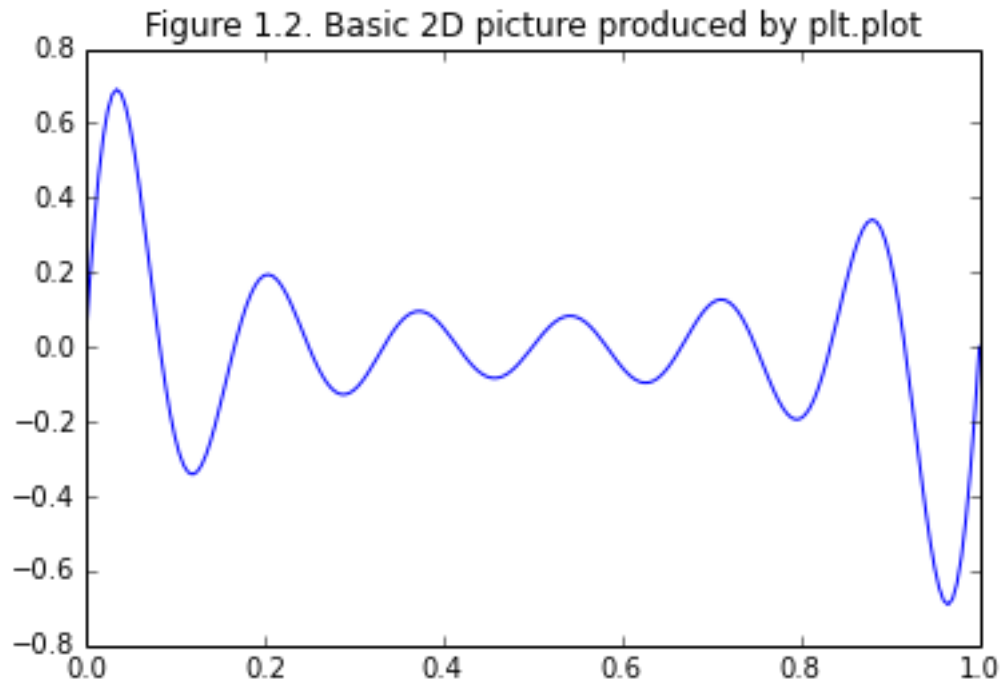
The plot function from the matplotlib module pylab produce two dimensional pictures. You can assess these plotting and graphics functions by importing this module. It is a common practice to rename it to plt, as in the following example.

```

In [50]: #import mpld3
         #mpld3.enable_notebook()
         import matplotlib.pyplot as plt
         t = np.arange(0, 1.005, 0.005)
         z = np.exp(10 * t * (t - 1)) * np.sin(12 * np.pi * t)
         plt.plot(t, z)
         plt.title("Figure 1.2. Basic 2D picture produced by plt.plot")

```

```
Out[50]: <matplotlib.text.Text at 0x6446dd8>
```

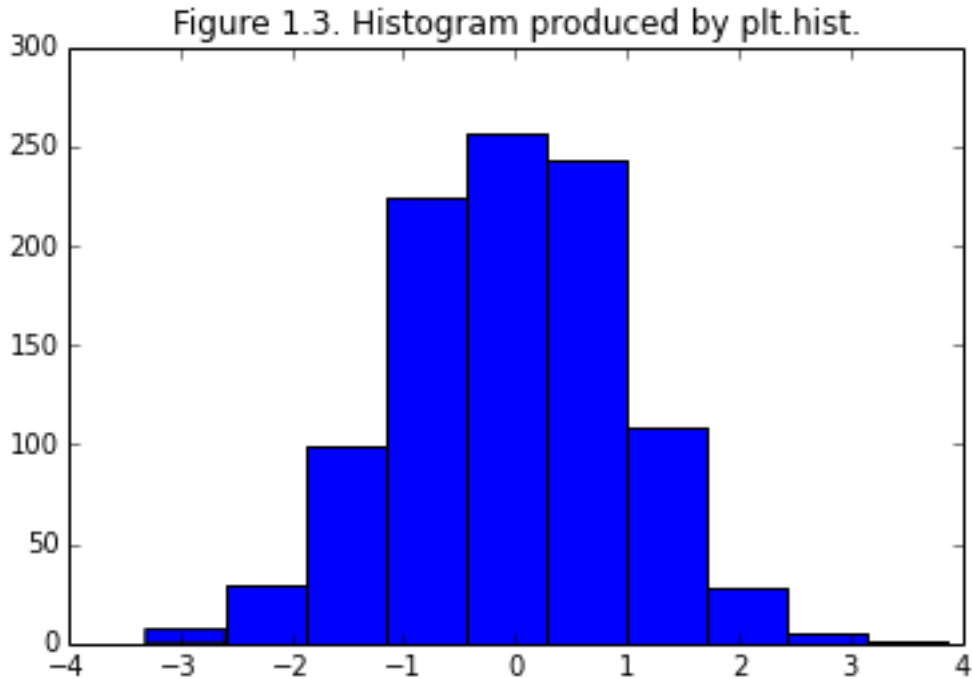


Here, `plt.plot(t, z)` joins the points `t[i]`, `z[i]` using the default solid linetype. Matplotlib opens a figure window in which the picture is displayed. In this IPython notebook, the matplotlib figures are included 'in line' since the notebook was invoked with the "ipython notebook --pylab=inline" (the `--pylab=inline` option will be deprecated in the 3.0 notebook in favor of the `%matplotlib inline` cell magic) command. When you are using IPython as a command line interpreter, after opening a plot figure window, you can close it in the normal way, ie by clicking on the x in the window title bar.

You can produce a histogram with the function `plt.hist`

```
In [51]: plt.hist(np.random.randn(1000))  
         plt.title("Figure 1.3. Histogram produced by plt.hist.")
```

```
Out[51]: <matplotlib.text.Text at 0x66687b8>
```



Here, hist is given 1000 points from the normal (0, 1) random number generator

You are now ready for more challenging computations. A random Fibonacci sequence  $\{x_n\}$  is generated by choosing  $x_1$  and  $x_2$  and setting

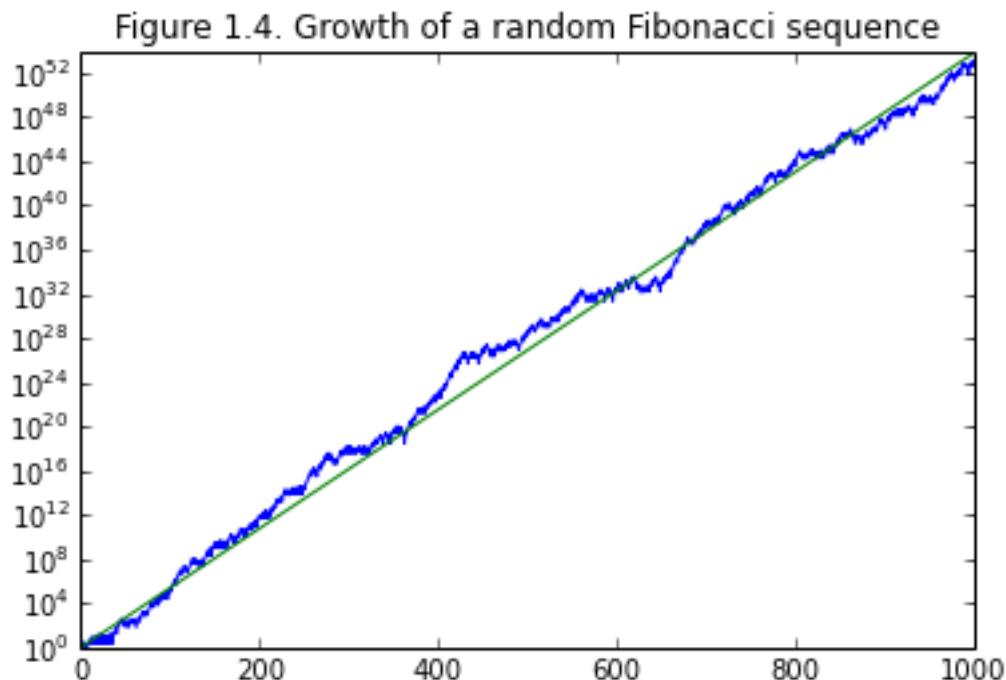
$$x_{n+1} = x_n \pm x_{n-1}, \quad n \geq 2$$

Here, the  $\pm$  indicates that  $+$  and  $-$  must have equal probability of being chosen. Viswanath [121] listed in [1] analyzed this recurrence and showed that, with probability 1, for large  $n$  the quantity  $|x_n|$  increases like a multiple of  $c^n$ , where  $c = 1.13198824\dots$  (see also [25]). You can test Viswanath's result as follows:

```
In [52]: %reset -f
import numpy as np
import pylab as plt
rd = np.random.RandomState()
rd.seed(100)
x = np.zeros(1000)
x[0:2] = 1, 2
for n in range(1, 999):
    x[n + 1] = x[n] + np.sign(rd.rand(1) - 0.5) * x[n - 1]

xx = np.arange(1, 1001)
plt.semilogy(xx, np.abs(x))
c = 1.13198824
plt.hold(True)
plt.semilogy(xx, c ** xx)
plt.title("Figure 1.4. Growth of a random Fibonacci sequence")
plt.hold(False)
```





Here, `%reset -f` removed all variables and imported modules from the workspace, which is why we needed to import `numpy` and `pylab` again. The for loop stores a random Fibonacci sequence in the array `x`; we pre-allocate `x` to the size needed and initial to zero using the `np.zeros()` function. The `plt.semilogy` function then plots `n` on the x-axis against `|X|` on the y-axis, with logarithmic scaling for the y-axis. Typing `np.hold(True)` tells matplotlib to superimpose the next picture on top of the current one. The second semilogy plot produces a line of slope `c`. The overall picture, shown in Figure 1.4, is consistent with Viswanath’s theory.

We can make the above code into a command by writing it out to a file. If you cut and paste it into a file called `fib.py`, you can then run it in IPython by typing “run fib” or “run fib.py” to reproduce the above graph.

However, you can experiment directly in this IPython notebook by changing any of the values and then hitting the ‘play’ button above.

Our next example involved the Collatz iteration, which, given a positive integer  $x_1$ , has the form  $x_{k+1} = f(x_k)$ , where

$$f(x) = \begin{cases} 3x + 1 & : x \% 2 == 0 \\ x/2 & : x \% 2 == 1 \end{cases}$$

Here  $x \% y$  is the modulus function of  $x$  and  $y$ , sometimes also written as `mod(x, y)`. It is the remainder of  $x$  when divided by  $y$ . In this case  $x \% 2 == 0$ , if true, means that  $x$  is even and if  $x \% 2 == 1$  is true, then  $x$  is odd. Note that these are perfectly good Python functions

```
In [53]: x = 3
         if x % 2 == 0:
             print ('x is even')
         else:
             print ('x is odd')
```

```
x is odd
```

But now returning to our quest, the equation above in words means: if  $x$  is odd, replace it by  $3x + 1$ , and if  $x$  is even, halve it. It has been conjectured that this iteration will always lead to a value of 1 (and hence

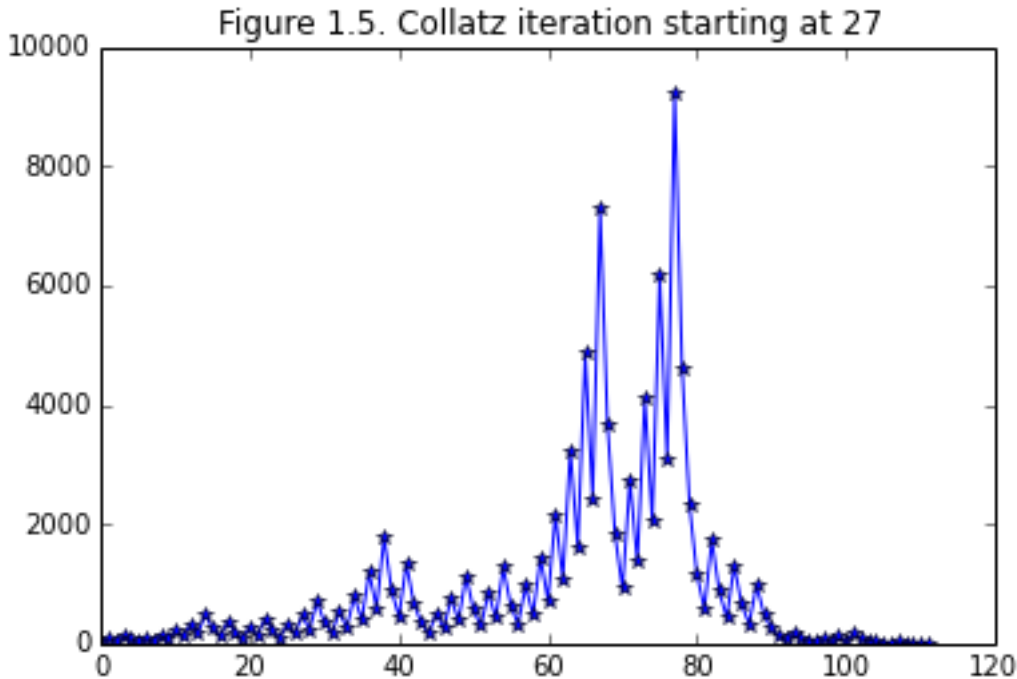
thereafter cycle between 4, 2, and 1) whatever starting value  $x_1$  is chosen. There is ample computational evidence to support this conjecture, which is variously known as the Collatz problem, the  $3x+1$  problem, the Syracuse problem, Kakutani's problem, Hasse's algorithm, and Ulam's problem. However, a rigorous proof has so far eluded mathematicians. For further details, see [63] listed in [1] or type "Collatz problem" into your favorite Web search engine. You can investigate the conjecture by creating the python script [U+FB01]le collatz.py shown below. In this file a while loop and an if statement are used to implement the iteration. You can run this in the notebook simply by changing the value of n and hitting the play button. Or, you can save this to a file named collatz.py (remembering to uncommment out the line '#n = int(raw\_input...)' and commenting out the line "n = 27". Then, from the IPython shell prompt, type "run collatz". The input command prompts you for a starting value. The appropriate response is to type an integer and then hit return.

In [54]: *#COLLATZ Collatz iteration.*

```
#n = int(raw_input('Enter an integer bigger than 2: '))
init_n = n = 27
narray = np.zeros(1000) # only a maximum of 1000 iterations
narray[0] = n
count = 0
while n != 1:
    if n % 2 == 1: # Remainder modulo 2.
        n = 3 * n + 1
    else:
        n = n / 2
    count += 1
    narray[count] = n # Store the current iterate

# Plot with * marker and solid line style.
# Only plot the non zero entries in narray
plt.plot(narray[narray != 0], '*-')
plt.title('Figure 1.5. Collatz iteration starting at %d' % init_n)
```

Out[54]: <matplotlib.text.Text at 0x646f630>



A couple of things you should note about the code above. This first is that it will fail if the number of iterations goes beyond 1000. For most of the inputs I have tried, it ‘converged’ in under 200 iterations. But you should try to break it! The second thing to note is that we only want to plot the non zero values. All the zero values at the end of the array add no new information and we want to drop them. That is easily accomplished by using the `narray[narray != 0]` syntax which selects only the non-zero values out of `narray`.

To investigate the Collatz problem further, the script `collbar` in the next listing plots a bar graph of the number of iterations required to reach the value 1, for starting Values 1,2,. . . ,29. The result is shown in Figure 1.6. For this picture, the function `plt.grid` adds grid lines that extend from the axis tick marks, while `plt.title`, `plt.xlabel`, and `plt.ylabel` add further information.

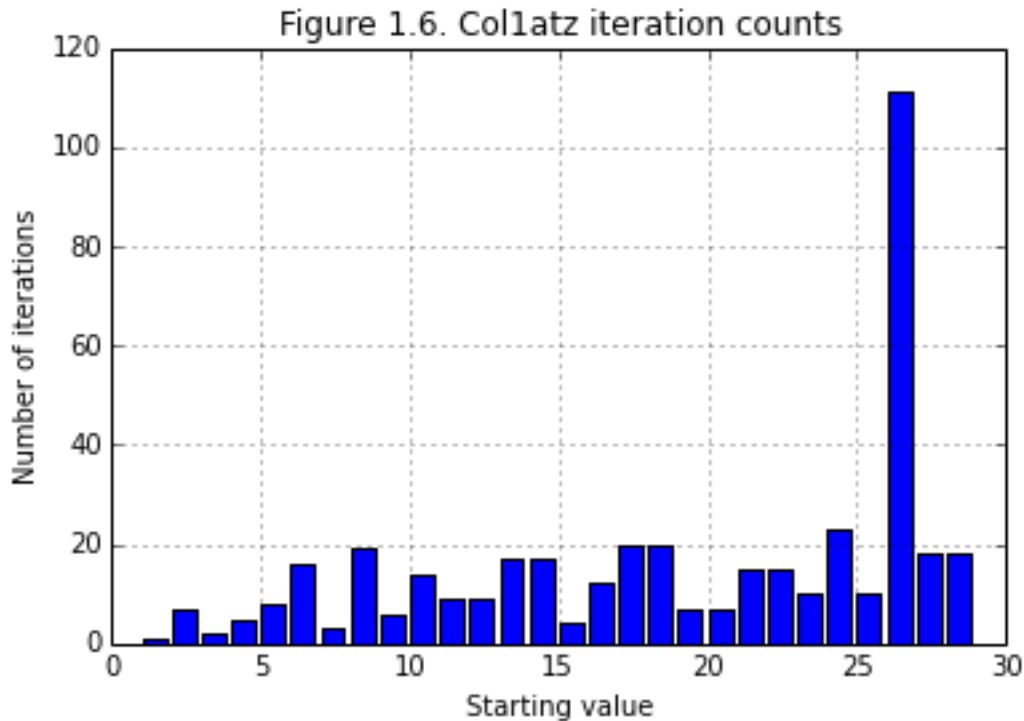
```
In [55]: #COLLBAR Collatz iteration bar graph.
N = 29 # Use starting values 1,2,...,N.
niter = np.zeros(N); # Preallocate array.
for i in range(N):
    count = 0
    n = i + 1
    while n != 1:
        if n % 2 == 1:
            n = 3 * n + 1
        else:
            n = n / 2

        count += 1
    niter[i] = count

left = np.arange(29)
plt.bar(left, niter) # Bar graph.
plt.grid() # Add horizontal and vertical grid lines.
plt.title('Figure 1.6. Collatz iteration counts')
```

```
plt.xlabel('Starting value') # Label X-axis.
plt.ylabel('Number of iterations') #Label y-axis.
```

Out[55]: <matplotlib.text.Text at 0x64af8d0>



The Well-known and much studied Mandelbrot set can be approximated graphically in just a few lines of Python/NumPy. It is defined as the set of points  $c$  in the complex plane for which the sequence generated by the map  $z \mapsto z^2 + c$ , starting with  $z = c$ , remains bounded [91, Chap. 14] listed in [1]. The script `mandel` in the next listing produces the plot of the Mandelbrot set shown in Figure 1.7. The script contains calls to `np.linspace` of the form `np.linspace(a, b, n)`, which generate an equally spaced vector of  $n$  values between  $a$  and  $b$ . The `meshgrid` and complex functions are used to construct a matrix  $C$  that represents the rectangular region of interest in the complex plane. The plot itself is produced by `plt.contourf`, which plots a filled contour. The expression `abs(Z) < Z_max` in the call to `contourf` detects points that have not exceeded the threshold `Z_max` and that are therefore assumed to lie in the Mandelbrot set; the `double` function is applied in order to convert the resulting logical array to numeric form. You can experiment with `mandel` by changing the region that is plotted, via the `linspace` calls, the number of iterations `it_max`, and the threshold `Z_max`. Also note that on some runs we may see a `RuntimeWarning` printed because we have an overflow on  $z$ . This can be silenced with a context manager but we will wait to introduce detailed control of errors and warnings.

In [56]: `#MANDEL Mandelbrot set.`

```
x = np.linspace(-2.1, 0.6, 301)
y = np.linspace(-1.1, 1.1, 301)
[X,Y] = np.meshgrid(x, y)
C = X + 1j * Y

Z_max = 1e6
```

```

it_max = 50
Z = C

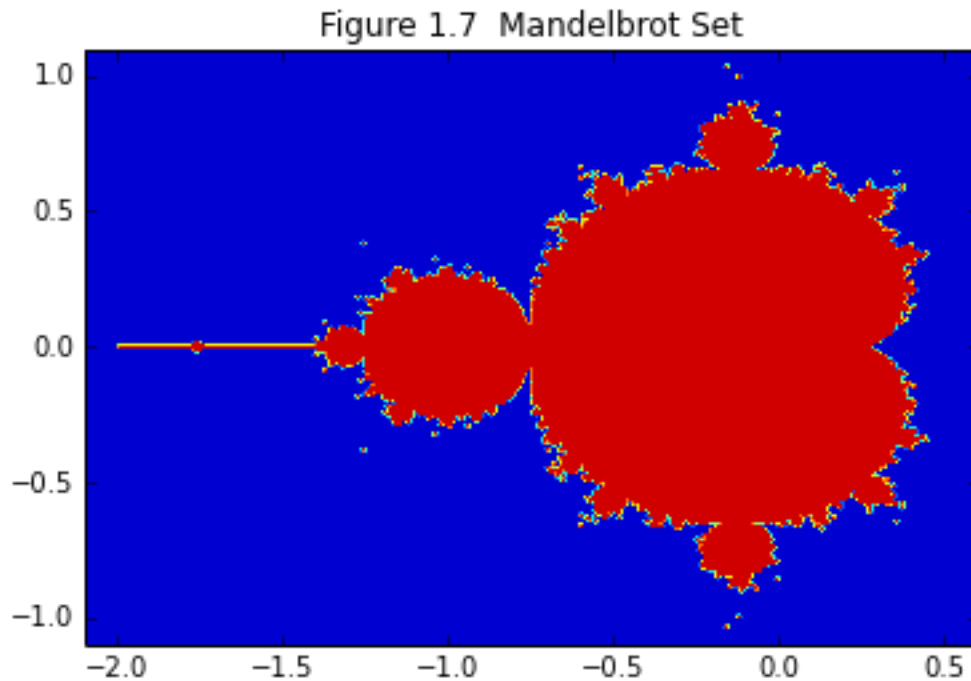
for k in range(it_max):
    Z = Z ** 2 + C

plt.contourf(x, y, np.float64(np.abs(Z) < Z_max))
plt.title("Figure 1.7 Mandelbrot Set")

```

-c:13: RuntimeWarning: overflow encountered in square  
-c:13: RuntimeWarning: invalid value encountered in square  
-c:15: RuntimeWarning: invalid value encountered in less

Out[56]: <matplotlib.text.Text at 0x65011d0>



Next we solve the ordinary differential equation (ODE) system

$$d/dt y_1(t) = 10(y_2(t) - y_1(t)),$$

$$d/dt y_2(t) = 28y_1(t) - y_2(t) - y_1(t)y_3(t),$$

$$d/dt y_3(t) = y_1(t)y_2(t) - 8y_3(t)/3.$$

This is an example from the Lorenz equations family; see [H1] listed in [1]. We take initial conditions  $y(0) = [0, 1, 0]$  and solve over  $0 \leq t \leq 50$ . The next listing, titled Lorenz, is an example of a Python function. Given  $t$  and  $y$ , this function returns the right-hand side of the ODE as the vector **yprime**. This is the form required by Scipy's ODE solving functions. The rest of the listing uses the scipy function odeint to solve the ODE numerically and then produces the  $(y_1, y_3)$  phase plane plot shown in Figure 1.8. You can try different values of the constants defining the derivatives to see what happens. For instance, change the 3 to an 8 in lorenzde and observe how the plot changes

```

In [57]: import scipy.integrate as si

def lorenzde(y, t):
    '''LORENZDE Lorenz equations.
       YPRIME = LORENZDE(Y,T)
    '''
    yprime = np.array([10. * (y[1] - y[0]), 28. * y[0] - y[1] - y[0] * y[2],
                       y[0] * y[1] - 8. * y[2] / 3.])
    return yprime

#lrun    ODE solving example: Lortez.

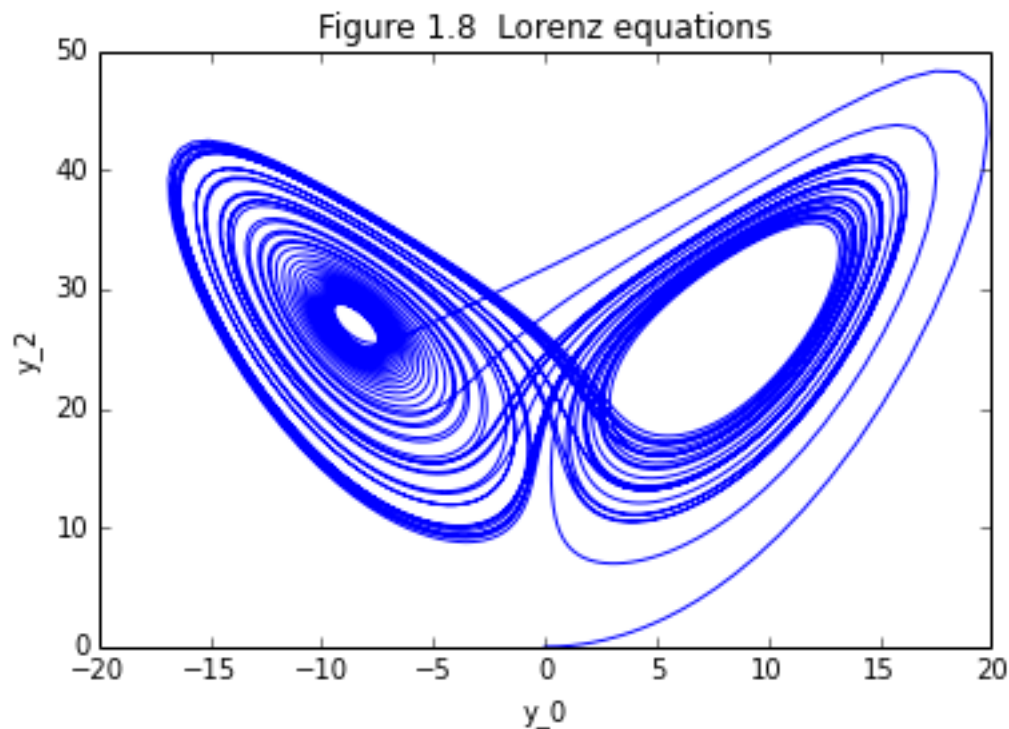
t = np.arange(0, 50.01, .01) # time points on which to solve
yzero = np.array([0., 1., 0.])
print (len(yzero))
y = si.odeint(lorenzde, yzero, t)

plt.plot(y[:, 0], y[:, 2])
plt.xlabel('y_0')
plt.ylabel('y_2')
plt.title('Figure 1.8 Lorenz equations')

```

3

Out[57]: <matplotlib.text.Text at 0x7d3c278>



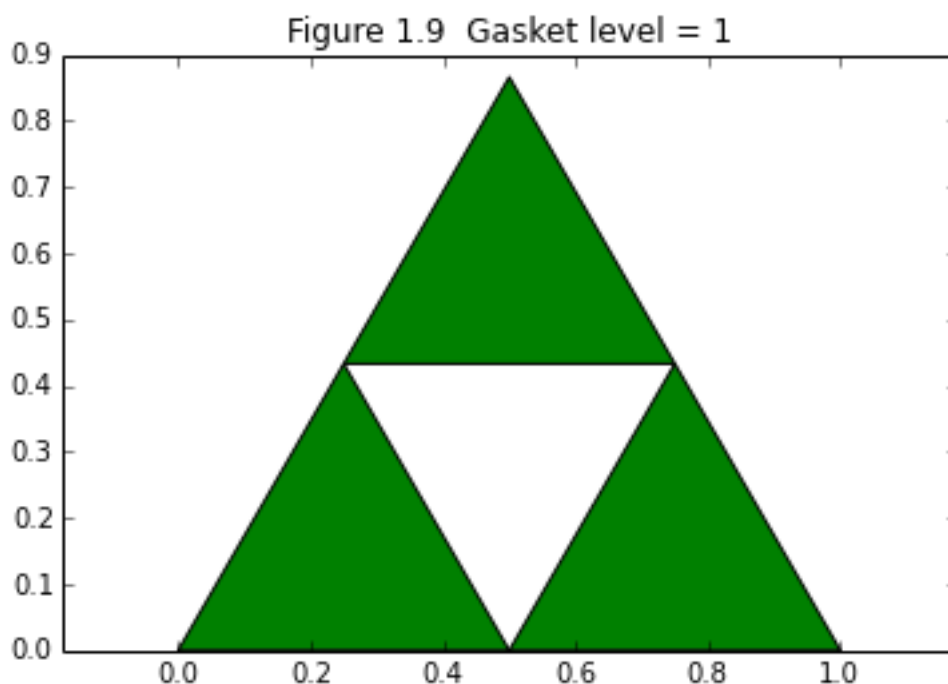
Now we give an example of a recursive function, that is, a function that calls itself. The Sierpinski gasket [90, Sec. 2.2] listed in [1] is based on the following process. Given a triangle with vertices  $P_a$ ,  $P_b$ , and  $P_c$ , We

remove the triangle with vertices at the midpoints of the edges,  $(P_a + P_b)/2$ ,  $(P_b + P_c)/2$ , and  $(P_c + P_a)/2$ . This removes the “middle quarter” of the triangle, as illustrated in Figure 1.9. (The code in the function ‘gasket’ will be explained below)

```
In [58]: def gasket(pa, pb, pc, level):
        '''
        GASKET Recursively generated Sierpinski gasket.
        GASKET(PA, PB, PC, LEVEL) generates an approximation to
        the Sierpinski gasket, where the 2-vectors PA, PB, and PC Z define the triangle vertices.
        LEVEL is the level of recursion.
        '''
        if level == 0:
            # Fill the triangle with vertices Pa, Pb, Pc.
            plt.fill([pa[0], pb[0], pc[0]], [pa[1], pb[1], pc[1]], 'g')
            plt.hold(True)
        else:
            # Recursive calls for the three subtriangles.
            gasket(pa, (pa + pb) / 2., (pa + pc) / 2., level - 1)
            gasket(pb, (pb + pa) / 2., (pb + pc) / 2., level - 1)
            gasket(pc, (pc + pa) / 2., (pc + pb) / 2., level - 1)

pa = np.array([0, 0])
pb = np.array([1, 0])
pc = np.array([0.5, np.sqrt(3)/2.])
level = 1
gasket(pa, pb, pc, level)
plt.hold(False)
plt.title("Figure 1.9 Gasket level = 1")
plt.axis('equal')
```

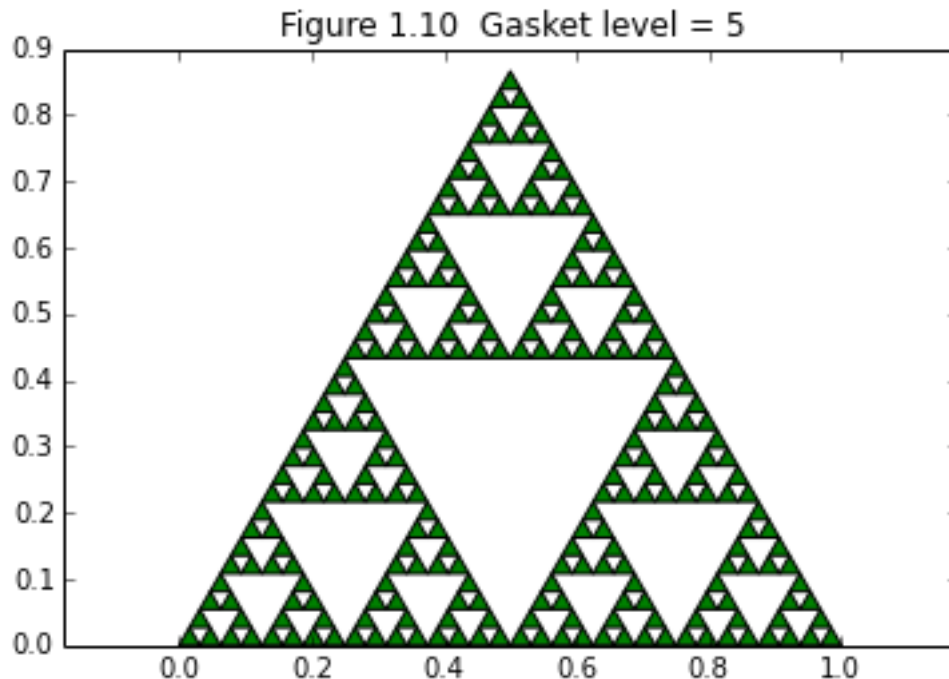
Out[58]: (0.0, 1.0, 0.0, 0.9000000000000002)



Effectively, we have replaced the original triangle with three “subtriangles”. We can now apply the middle quarter removal process to each of these subtriangles to generate nine subsubtriangles, and so on. The Sierpinski gasket is the set of all points that are never removed by repeated application of this process. The function `gasket` in the listing above implements the removal process. The input arguments  $P_a$ ,  $P_b$ , and  $P_c$  define the vertices of the triangle and `level` specifies how many times the process is to be applied. If `level` is nonzero then `gasket` calls itself three times with `level` reduced by 1, once for each of the three subtriangles. When `level` finally reaches zero, the recursion ‘bottoms-out’ and the appropriate triangle is drawn. The following code generates Figure 1.10.

```
In [59]: level = 5
         gasket(pa, pb, pc, level)
         plt.hold(False)
         plt.title("Figure 1.10 Gasket level = 5")
         plt.axis('equal')
```

```
Out[59]: (0.0, 1.0, 0.0, 0.90000000000000002)
```



(Figure 1.9 was generated in the same way with `level = 1`.) In the last line, the call to `axis` makes the units of the  $x$ - and  $y$ -axes equal and turns off the axes and their labels. You should experiment with different initial Vertices  $P_a$ ,  $P_b$ , and  $P_c$ , and different levels of recursion, but keep in mind that setting `level` bigger than 8 may overstretch either your patience or your computer’s resources.

The Sierpinski gasket can also be generated by playing Barnsley’s “chaos game” [90, Sec. 1.3] listed in [1]. We choose one of the vertices of a triangle as a starting point. Then we pick one of the three vertices at random, take the midpoint of the line joining this vertex with the starting point and plot this new point. Then we take the midpoint of the line joining this point and a randomly chosen vertex as the next point, which is plotted, and the process continues. The script `barnsley` in Listing 1.8 implements the game. Figure 1.11 shows the result of choosing 1000 iterations:



In [60]: *#BARNSELEY Barnsley's game to compute Sierpinski gasket.*

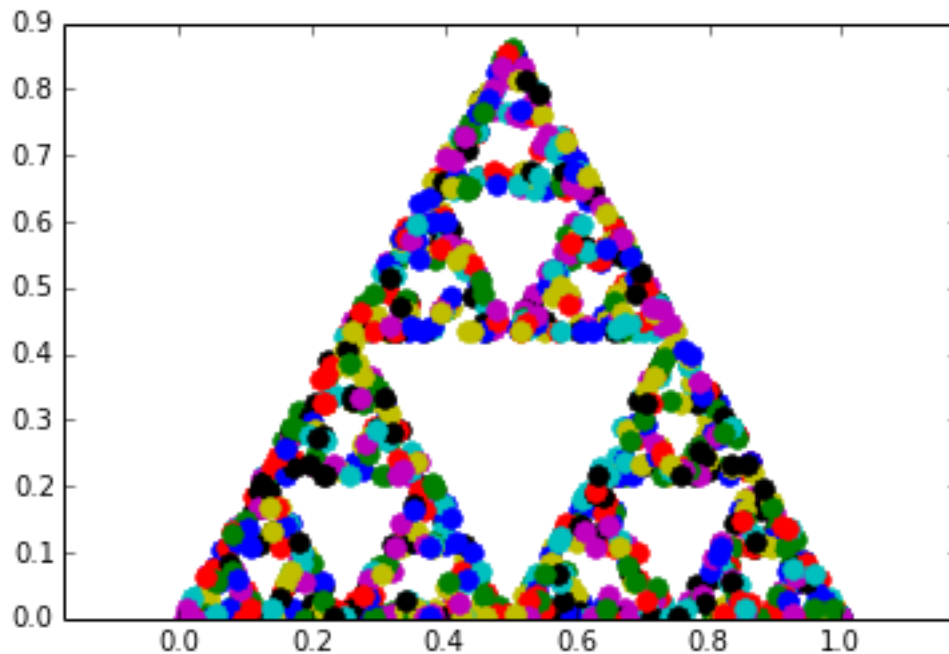
```
rd = np.random.RandomState() # Initialize a new RandomState object
rd.seed(1)                  # and (re)set the seed

V = np.c_[[0, 0], [1, 0], [0.5, np.sqrt(3)/2]] # Columns give triangle vertices.
point = V[:, 0]               # Start at a Vertex.

n = 1000                     # Change this number to experiment

for k in range(n):
    node = int(np.ceil(3 * np.random.rand()) - 1) # node is 0, 1, or 2 with equal prob.
    point = (V[:, node] + point)/2;
    plt.plot(point[0], point[1], ".", markersize=15)
    plt.hold(True)

plt.axis('equal')
plt.hold(False)
```



Try experimenting with the number of points,  $n$ , the type and size of marker in the plot command, and the location of the starting point.

We [U+FB01]nish with the listing of sweep, which generates a volume—swept three—dimensional (3D) object; see Figure 1.12. Here, the command `surf (X,Y,Z)` creates a 3D surface where the height  $Z[i, j]$  is speci[U+FB01]ed at the point  $(X[i, j], Y[i, j])$  in the x-y plane. The script is not written in the most obvious fashion, which would use two nested for loops. Instead it is vectorized. To understand how it works you will need to be familiar with Chapter 5 and 21.4 of reference [1]. You can experiment with the script by changing the parameter  $N$  and the function that determines the variable radius: try replacing `sqrt` by other functions, such as `log`, `sin`, or `abs`.

In [61]: *# SWEEP Generates a volume-swept 3D object.*  
`import numpy as np`

```

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')

n = 10      # Number of increments - try increasing

zz = np.linspace(-5, 5, n).reshape(n, 1)
radius = np.sqrt(1 + zz ** 2)    # Try changing sqrt to cos, sin, log or abs
theta = 2 * np.pi * np.linspace(0, 1, n)
x = radius * np.cos(theta)
y = radius * np.sin(theta)
z = zz[:, n * [0]]    # Tony's trick! Who is Tony? No idea.

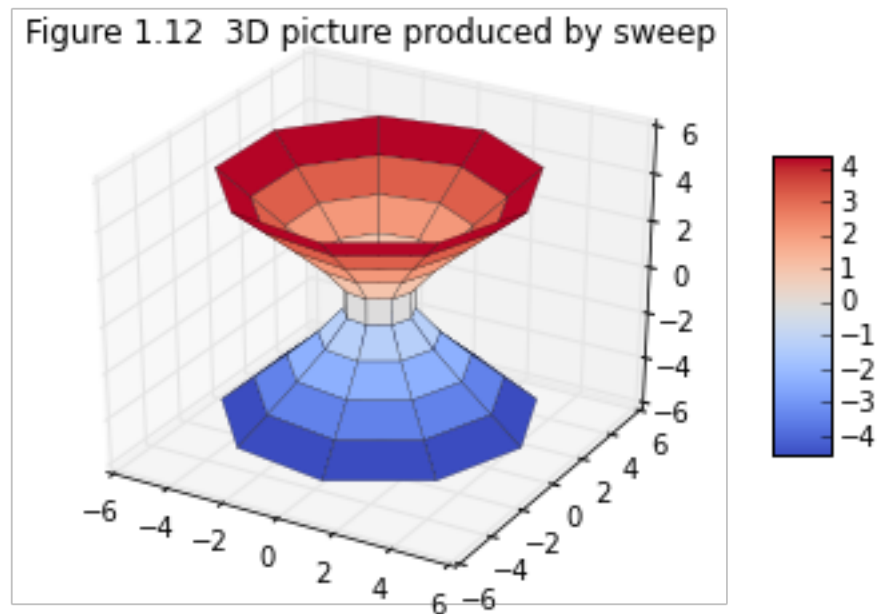
surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap=cm.coolwarm,
                      linewidth=.2, antialiased=True)

fig.colorbar(surf, shrink=0.5, aspect=5)

plt.title('Figure 1.12 3D picture produced by sweep')

```

Out[61]: <matplotlib.text.Text at 0xf667e48>



In [62]: cd

C:\Users\Elias

[1] D. J. Higham and N. J. Higham.  
[MATLAB Guide](#), Second edition,

Society for Industrial and Applied Mathematics, Philadelphia, PA, USA,  
2005, ISBN 0-89871-578-4

[2] Fernando Pérez, Brian E. Granger, IPython: A System for Interactive Scientific Computing, *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. URL: <http://ipython.org>