An Introduction to Python for Engineering Students

Mike Perkins
Adjunct Professor, Carnegie Mellon University Africa

January 10, 2022

1 Introduction

Python is a powerful scripting language whose basics can be easily mastered by anyone with programming experience. One of its primary advantages is that it is supported by a large number of libraries providing specialized functionality such as data analysis, mathematical programming, and plotting. Python is used extensively in industry for a diverse range of applications. For many engineering applications, it is a great free alternative to Matlab.

This document assumes the reader has programming experience in some language. The basics of computer science are not covered. The goal is to help an engineer quickly pick up the Python features needed to start doing technical programming. No attempt is made to provide a complete presentation of Python's features. The following sections are mostly a series of self-explanatory scripts demonstrating how to perform many of the tasks an engineer might need to do.

Finally, I do not claim that the examples shown are Python best practice. I come from a "C/C++" programming background, so you'll detect a bias towards "C"-type thinking in my examples. This document is based on Python 3.7.

Some links to other material you may find useful can be found in Section 11.

2 High Level Program Structure

Lets assume you need to write a program that will use two helper functions called by a main routine. A workable structure for such a program is the following:

```
#!/usr/local/bin/python3

def myFunctionOne(x, y) :
  # Do some processing here
  w = x + y
  z = x * y
  return w, z

def myFunctionTwo(x, y) :
  w = x - y
  z = x / y
  return w, z

############
# Here is our main block of code.  Execution starts here
############
a = 5.0
b = 6.0

ret1, ret2 = myFunctionOne( a, b )
print( "ret1 = ", ret1, " ret2 = ", ret2 )

ret1, ret2 = myFunctionTwo( a, b )
print(f"ret1 = {ret1}  ret2 = {ret2}")
```

The first line informs the computer how to run Python, i.e., where to find the python executable. You made need to change it based on how Python was installed on your computer.

Each function begins with a `def` keyword followed by the name of the function and the function's arguments in parentheses. Note the colon at the end of the definition line. It's mandatory. Note as well that the lines of code belonging to the function are indented. Indentation matters in Python! The Python interpreter uses the indentation level to know what code is nested under what higher-level structure. Without proper indentation, your script will not run correctly! So pay attention to indentation and make sure that lines of code belonging at the same nesting level line

up.

Comments are lines that begin with a `#`. Finally, note that both these functions return two values instead of just one. The main code block captures both return values and then prints them. Two ways to print variables are shown.

If you prefer to separate your functions and the main block of code into two files it can easily be done. Just put your functions into a separate file with a name that ends in `.py`. I used the name `examp02a.py` for my module file and its contents look like this:

```
def myFunctionOne(x, y) :
  # Do some processing here
  w = x + y
  z = x * y
  return w, z

def myFunctionTwo(x, y) :
  w = x - y
  z = x / y
  return w, z
```

Then put your main code block into a second file whose name also ends in `.py`, and add an import line for your module as shown below:

```
#!/usr/local/bin/python3

import examp02a as mf

############
# Here is our main block of code
############
a = 5.0
b = 6.0

ret1, ret2 = mf.myFunctionOne( a, b )
print( "ret1 = ", ret1, " ret2 = ", ret2 )

ret1, ret2 = mf.myFunctionTwo( a, b )
print(f"ret1 = {ret1} ret2 = {ret2}")
```

When you execute the main code block (for example, from a shell prompt just type

`./filename.py` after ensuring the file has execute permission), the module containing your functions is imported. Note that to avoid typing the entire module name whenever we want to access one of the module's functions, we give it the shortcut name `mf`. We can then access the module's functions using the `mf.` syntax as shown.

It's natural to ask whether Python passes variables "by value" or "by reference". It turns out for Python this is a fairly complicated issue. Python's unit for data abstraction is an object and every object has 3 characteristics: an address, a type, and a value. Types can either be mutable or immutable. Immutable variable types pass by value—these include scalar variables like ints, floats, bools, and strings. Mutable types pass by reference—these include lists and arrays. We'll get into this more in some of the later examples. In the example above, the function's argument types are immutable so they are passed by value. In other words, changes to them inside a function are not seen by the calling function. You can always determine a variable's type with the line `print( type(variable) )` in your script.

Note that so far we have not declared any variable types. Python figures out what variable type to use based on how your code uses it. If we had written

```
a = 5
b = 6
```

instead of

```
a = 5.0
b = 6.0
```

Python would have treated a and b as integers instead of floats.

Note that it is also possible to assign several variables the same value in a single line, for example

```
a = b = 5.0
```

3 For Loops and If/Then/Else Statements

The following example shows the use of two common control structures, as well as a simple case of inputting a value from the terminal.

```
#!/usr/local/bin/python3

print( "Enter a number between 1 and 10" )
str_value = input( )
int_value = int( str_value )
print( "you entered ", int_value )

if int_value > 10 :
  print( "The number you entered is too large" )
elif int_value < 1 :
  print( "the number you entered is too small" )
else :
  for i in range(int_value) :
    print( i )
```

Starting at the top, we see that the function `input( )` returns a string value from the terminal. We convert this string value to an integer so we can use it as such in the `for` loop. The conversion resembles casting in "C". Other casting operations are also supported by Python, for example, strings can be converted to floats using `float( )` and integers and floats can be converted to strings using `str( )`.

The rest of the example is self-explanatory except for the `range( )` function. `range(` N `)` generates the values 0 to $N-1$ inclusive. `range( )` can optionally be invoked as `range(start_val, upper_val, increment)`. In this case, on the first pass through the loop the loop variable assumes the value `start_val`; the loop variable then increments by `increment` each pass until it exits; the loop exits without executing when an increment causes the loop variable to equal or exceed `upper_val`. If the increment size is left unspecified, it defaults to 1, so `range(3, 8)` will generate the loop variable values 3 through 7 inclusive.

4 Reading Text Data From a File

Let's say we have a file with several words on each line. In other words, each line has a simple sentence like "Joe hit the ball" or "Jane ate a burrito". We want to read each line, determine how many words are on it, count the total number of words in the file, and build a list of the first word that appears on each line. The following code will do the trick:

```
#!/usr/local/bin/python3
```

```
print( "input the file name" )
filename = input( )
print( "you entered the name ", filename )

fd = open( filename, 'r' )

num_words = 0
first_words = []
for line in fd :
  words = line.split()
  num_words += len( words )
  first_words.append( words[0] )

print( num_words )
print( first_words )
fd.close()
```

This is a bit more complicated than the earlier examples because we're starting to use more of Python's power. The first new thing is opening a file for reading (later we'll see how to catch errors). Then we initialize two variables: `num_words` is an integer (note we used 0 not 0.0) and we use it to count the total number of words in the file; `first_words` is declared as an empty list and we'll fill it up with the first word from every line. The `for( )` loop reads one line at a time from the file and puts each line in the string variable `line`. We then use the function `split( )`—inherent to the string class—to split the line into a new list named `words`. Once this is done we use the function `len( )` to determine how many words are in the `words` list and we accumulate the total number of words in `num_words`. We then use the list function `append( )` to add the first word from the `words` list to the list `first_words` (lists are indexed and indexing starts at 0). Finally, we print out our results and close the file we opened.

5 Writing Text Data to a File

Writing text data to files is also easy as demonstrated in the following example.

```
#!/usr/local/bin/python3
import sys
```

```
if len( sys.argv ) != 3 :
  print( "Usage is: %s file_to_open num_lines_to_write" % (sys.argv[0]) )
  sys.exit()

fd = open( sys.argv[1], 'w' )
lines2write = int( sys.argv[2] )

for i in range( lines2write ) :
  print( "Enter a line of data for the file" )
  print( "each line should be two words followed \
by a floating point number")
  line = input( )
  words = line.split()
  fd.write('%s %s %f %8.2f\n' % \
  (words[0], words[1], float(words[2]), float(words[2]) )  )

fd.close()
```

As for new things in this example, note that we've included the standard Python module `sys`. One thing this module enables is the capturing of command line arguments when a script is invoked. The entire command line is available inside the script via the list `sys.argv`. In this example, we first check to ensure that two arguments appear after the script's name (remember: the entire command line appears in `sys.argv`, so `sys.argv[0]` always contains the name of the script currently executing). If the script is incorrectly invoked, we print a useful message for the user and exit. Note the use of the `exit()` function which is also in the `sys` module. The output file is then opened for writing—its name is the first command line argument—and the number of lines of data we will type into the file is converted to an integer.

Inside the `for( )` loop we perform a formatted output to the file. Note that we used the line continuation character "\" for a long line that we want to wrap around to the next line in the script. The backslash simply causes the newline following it to be ignored. A "C"-like syntax is used for writing formatted output to the file. String variables are denoted by `%s` and floating point numbers by `%f`. If we had wanted to print out an integer, we would have used `%d`. Note that we do some casting to floats in this example. Controlling the width of a field and the number of decimal points for a float is also demonstrated for the last value output on each line. Finally, the file is closed.

Note the following is an alternative way to control the output formatting which you may find easier to remember (look at the 2nd to last line below).

```
#!/usr/local/bin/python3
import sys

if len( sys.argv ) != 3 :
  print( "Usage is: %s file_to_open num_lines_to_write" % (sys.argv[0]) )
  sys.exit()

fd = open( sys.argv[1], 'w' )
lines2write = int( sys.argv[2] )

for i in range( lines2write ) :
  print( "Enter a line of data for the file" )
  print( "each line should be two words followed \
by a floating point number")
  line = input( )
  words = line.split()
  fd.write(f'{words[0]:15s} {words[1]:10s} {float(words[2]):8.3f}\n')
fd.close()
```

6 1-D Arrays and Binary Input and Output

First we'll discuss a script that outputs to a binary file. We'll start using the numpy module now, so if you don't have it installed yet, go get that package.

```
#!/usr/local/bin/python3
import sys
import numpy as np

if len( sys.argv ) != 2 :
  print( "Usage is: %s file_to_write" % (sys.argv[0]) );
  sys.exit()

fd = open( sys.argv[1], 'wb' )

# Declare, initialize, then output a byte array
# Note the compact initialization of a standard python
```

```
# list which is then converted to a numpy array of type uint8
myData = np.asarray( [i for i in range(256)], np.uint8 )
print(f"myData = {myData}")
myData.tofile(fd, "")

# Below demonstrates how to write other
# "C"-like variables to a file.
# Note that this is conceptually just casting

myCint = np.int32(-82*45)
myCint.tofile(fd)

myCuint = np.uint32(82*45)
myCuint.tofile(fd)

myCdouble = np.float64(25.87 / 34.76)
myCdouble.tofile(fd)

fd.close()
```

First, because we are opening a binary file, note that the permissions are now `'wb'` instead of just `'w'`. Next, note how an array of type uint8 is compactly created and initialized to the numbers 0 to 255. This example uses python's standard list initialization capability, and the resulting python list is then turned into a numpy array. The script goes on to demonstrate how individual array members are accessed using [ ] and how the contents of the entire array can be written in binary form to a file. Numpy makes available a full assortment of data types including `int8, uint8, int16, uint16, int32, uint32, int64, uint64, float16, float32, float64`, `complex64`, and `complex128`. (It is worth noting that python has its own array types, but we will only use numpy arrays for math and signal processing. Also, keep in mind that python lists and numpy arrays are not the same thing)

One thing tricky about Python for those used to strongly typed languages is how the type currently associated with a variable name can change within a script. When in doubt, you can always force a type with a cast of the form `np.type( )` using one of the types listed above. As an example of this issue, the following code will yield unexpected results for those used to thinking like a "C" programmer.

```
#!/usr/local/bin/python3
import numpy as np
```

```
x = np.uint8( 2 )
print( type(x) )
x = x * 16
print( type(x) )
x = 32
print( type(x) )
```

If you run this script you will see that the first print statement indicates that x is a `numpy.uint8` as expected. The second print statement will indicate that x is a `numpy.int64` —even though the number 32 will easily fit into a `uint8`. Finally, the third print statement will indicate that x is an `int`. The initial cast to a `uint8` clearly does NOT declare x to be a fixed type for the duration of the function!

Numpy provides a way to declare an array of values in a manner similar to `malloc` for those familiar with 'C', and the type of the elements of this array will not be changed by math operations. In the example below, an array of size 10, initialized to all zeros, is created. The numpy function `np.ones()` could have been used to create an array initialized to all 1's instead. The code operations on elements of this numpy array will preserve the type `uint8` as math is performed:

```
#!/usr/local/bin/python3
import numpy as np

x = np.zeros(10, np.uint8 )
print( type(x[0]) )
x[0] = x[0] * 16
print( type(x[0]) )
x[0] = 320
print( type(x[0]) )
```

To demonstrate binary input we'll read the file output by the earlier binary output example:

```
#!/usr/local/bin/python3
import sys
import numpy as np

if len( sys.argv ) != 2 :
  print( "Usage is: %s file_to_read" % (sys.argv[0]) )
  sys.exit()

fd = open( sys.argv[1], 'rb' )
```

```
myData = np.fromfile(fd, np.uint8, 256)

for i in range(256) :
  if myData[i] != i :
    print( "Error occurred on uint8 value %d" % (i) )

myCint = np.fromfile(fd, np.int32, 1)
if myCint[0] != -82 * 45 :
  print( "Oops: an error occurred on the uint32" )

myCuint = np.fromfile(fd, np.uint32, 1)
if myCuint[0] != 82 * 45 :
  print( "Oops: an error occurred on the uint32" )

myCdouble = np.fromfile(fd, np.float64, 1)
if myCdouble[0] != 25.87 / 34.76 :
  print( "Oops: an error occurred on the float64" )

fd.close()
```

Observe that we used the permission string `'rb'` when opening the file for reading binary data. Data is read into an array using the method `fromfile()` where the first argument is the file designator returned by the call to `open( )`, the second argument is the data type being read, and the third argument is the number of items of this data type to read.

7 Python byte strings

Next we'll consider python byte strings, which are just raw strings of bytes. They are easy to initialize and easy to manipulate. Byte strings can be indexed, byte by byte, just like any other python list: the result is the value of that byte as an `int`. All numpy arrays and variables can be easily converted to and from byte strings. It is also easy to convert regular python strings to and from `utf-8` and `ascii` encoded byte strings.

The script below illustrates how several numpy arrays and variables can be converted to and from byte strings. The individual bytes are output to a file and then read

back from it. Perhaps the most important thing to observe below is the syntax of the `frombuffer()` command. Its arguments are the byte string to convert, the datatype, and the number of items to convert.

```
#!/usr/local/bin/python3
import sys
import numpy as np

if len( sys.argv ) != 2 :
  print( "Usage is: %s file_to_write" % (sys.argv[0]) );
  sys.exit()

fd = open( sys.argv[1], 'wb' )

# create some binary data
a = np.asarray( [i for i in range(256)], np.uint8 )
b = np.int32(-82*45)
c = np.asarray([1, 2, 3, 4], np.uint32)
d = np.asarray( [ 25.87 / 34.76, 1.2345], np.float64)

# output some binary data
fd.write(a.tobytes())
fd.write(a[123].tobytes())

fd.write(b.tobytes())

fd.write(c.tobytes())
fd.write(c[2].tobytes())

fd.write(d.tobytes())
fd.write(d[0].tobytes())

fd.close()

# read in all the data as a python b-string, then
# step through it converting it into numpy format
fd1 = open(sys.argv[1], 'rb')
data = fd1.read()
print(f"type(data) = {type(data)}")

buf_start = 0
```

```
buf_end = buf_start + 256 * np.dtype(np.uint8).itemsize
a1 = np.frombuffer( data[buf_start:buf_end], np.uint8, 256 )

buf_start = buf_end
buf_end = buf_start + 1 * np.dtype(np.uint8).itemsize
ap = np.frombuffer( data[buf_start:buf_end], np.uint8, 1)

buf_start = buf_end
buf_end = buf_start + 1 * np.dtype(np.int32).itemsize
bp = np.frombuffer( data[buf_start:buf_end], np.int32, 1)[0]

buf_start = buf_end
buf_end = buf_start + 4 * np.dtype(np.uint32).itemsize
c1 = np.frombuffer( data[buf_start:buf_end], np.uint32, 4)

buf_start = buf_end
buf_end = buf_start + 1 * np.dtype(np.uint32).itemsize
cp = np.frombuffer( data[buf_start:buf_end], np.uint32, 1 )

buf_start = buf_end
buf_end = buf_start + 2 * np.dtype(np.float64).itemsize
d1 = np.frombuffer( data[buf_start:buf_end], np.float64, 2 )

buf_start = buf_end
buf_end = buf_start + 1 * np.dtype(np.float64).itemsize
dp = np.frombuffer( data[buf_start:buf_end], np.float64, 1)[0]

# Check for an error
if np.array_equal(a, a1) == False : print("ERROR 1")
if a[123] != ap : print("ERROR 2")
if b != bp : print("ERROR 3")
if np.array_equal(c, c1) == False : print("ERROR 4")
if  c[2] != cp : print("ERROR 5")
if np.array_equal(d, d1) == False : print("ERROR 6")
if  d[0] != dp : print("ERROR 7")
```

Byte strings are easy to create by typing them in as ascii characters. They can also be entered in hex format. The script below shows some useful operations.

```
#!/usr/local/bin/python3
import sys
```

```
import os
import numpy as np

# create a b string and print it.  Note the leading b
a = b'how are you today'
print(f"a = {a}\n")

# Now print out the string's bytes in hex in three
# different formats. Note the use of "-1" as the 3rd argument
# to frombuffer().  This forces the entire array to be
# processed without having to explicitly give its length
print(a.hex())
print( ":".join("{:02x}".format(c) for c in a ) )
vhex = np.vectorize(hex)
print( vhex( np.frombuffer(a, np.uint8, -1) ) )

# b-strings can be specified byte-by-byte as well.
# Here's a string for the Euro sign
a = b'\xe2\x82\xac'
print(f"a = {a.decode('utf-8')}")

# Convert a regular python string to byte strings
# in unicode and ascii format
a = "today is a great day"
b = a.encode('utf-8')
c = a.encode('ascii')

# Convert byte strings back into regular python strings
print( b.decode('utf-8') )
print( c.decode('ascii') )
```

Finally, you might need to access the individual bits of a byte string. This script demonstrates how it can be done by unpacking the bits of a byte into a `uint8` array. A demonstration of how to perform bitwise logical operations at the byte level is also provided.

```
#!/usr/local/bin/python3
import numpy as np
import sys
import math
import os
```

```
# Create a np.uint8 array of 1's and 0's and pack it into bits
# Use the pattern 0xa0f3
dat = np.asarray([1,0,1,0, 0,0,0,0, 1,1,1,1, 0,0,1,1], np.uint8)
dat_packed = np.packbits(dat)

#Vectorize the built-in function hex for nice printing, verify the packing
vhex = np.vectorize(hex)
print(f"dat_packed = {vhex(dat_packed)}")

# convert the packed data back to an np.uint8 array and print it
dat_recon = np.unpackbits( dat_packed )
print(f"dat_recon = {dat_recon}")

# Input a file as a b-string, then convert it to a np.uint8 array
# that permits modification ("np.copy" is mandatory fo allow
# modifications!!!).  Then convert it back to a
# b-string and output it
fdw = open('myfile.dat', 'wb')
fdw.write(b'how are you today'); fdw.close()
fdr = open('myfile.dat', 'rb')
indat_b = fdr.read() ; fdr.close()
indat = np.copy( np.frombuffer(indat_b, np.uint8, -1) )

# manipulate indat as desired, in this case EOR
# some bytes together, then output the array
indat[0] = np.bitwise_xor( indat[1], indat[2] )
fdw = open('mynewfile.dat', 'wb')
fdw.write( indat.tobytes() ); fdw.close()
```

8 Higher Dimensional Arrays

In many cases, including image processing, it is useful to have higher-dimensional arrays. The script below demonstrates the use of 3-D arrays; higher dimensions are handled analogously.

```
#!/usr/local/bin/python3
import numpy as np
import sys
```

```
sys.path.append("/Users/perk/perkpylib/")
print( sys.path )
import bmp_io_perk

rows = 300
cols = 255
color_planes = 3

mypic = np.zeros( (rows, cols, color_planes), np.uint8)
print( "mypic.shape = ", mypic.shape )
print( "rows = ", rows, "mypic.shape[0] = ", mypic.shape[0] )
print( "cols = ", cols, "mypic.shape[1] = ", mypic.shape[1] )
print( "color_planes = ", color_planes, "mypic.shape[2] = ", \
                                    mypic.shape[2] )

# Create an image ramp by filling in the R, G, and B color planes with
# the same value (this makes a grey scale image).  Note the efficient
# way to do this with numpy using slicing
"""
for i in range(rows):
  for j in range(cols):
    mypic[i, j, 0] = mypic[i, j, 1] = mypic[i, j, 2] = j
"""
for j in range(cols):
  mypic[:, j] = j

# Output the image and read it back into a different array
bmp_io_perk.output_bmp("ramp.bmp", mypic);
rows, cols, mypic1 = bmp_io_perk.input_bmp("ramp.bmp")
bmp_io_perk.output_bmp("ramp_cmp.bmp", mypic1);

# Change one pixel: Note the efficient way to do this
"""
for k in range(color_planes):
  mypic1[100, 25, k] = 0
"""
mypic1[100, 25] = 0

# See if the two arrays are equal: use numpy
# to do this efficiently
if np.array_equal(mypic, mypic1) == False:
```

```
  print("The two images are different")

# Look for the location of the pixel we changed
# Note the efficient implementation using np.where
"""
for i in range(rows):
  for j in range(cols):
      if np.array_equal(mypic[i, j], mypic1[i, j]) == False :
        print( "pixel values differ at location [%d, %d]" % (i, j) )
"""
print("Locations where arrays differ=")
print(np.where(mypic != mypic1))
```

In this example we assume that the user has a module named `bmp_io_perk.py` in a user-defined directory. We append this directory name to the list variable sys.path so that when import is called, it too will be searched for modules. The `bmp_io_perk` module is assumed to contain two functions: one for outputting a 3-D array of `uint8's` to a BMP file, and one for inputting a BMP file, and putting its pixels into a 3-D array of `uint8's`. The construct for declaring the 3-D array is a straightforward extension of that used for a 1-D array. Array indexing is also analogous. Note also that the `shape` of an array can be determined as shown, and that the shape list can be accessed to determine the size of each array dimension (e.g. `rows` equals `mypic.shape[0]`).

This script also demonstrates a couple of extremely useful features of python indexing. First, it shows that it is not necessary to list every index. For example, all three color plane values were set to the same value, 0, with a single line of code: `mypic1[100, 25] = 0`. As a further example, when creating the ramp picture, the third index was dropped, while the `":"` in the row index location says apply the operation to every row index value. As a result, each column's three color plane values are set to `j` (the relevant line of code is: `mypic[:,j] = j`). Finally, block comments were created in a non-python-endorsed way using `"""` to create multi-line strings. Often your editor will handle block comments efficiently for you, but if you use an old-school editor, like vi, this is useful (and unlikely to get you into trouble).

The short script below shows how arrays can be initialized with hard-coded values. Note how the type of values stored in an array is controlled by the second argument to `np.asarray()`. This script also demonstrates how the print precision of floating point numbers can be set. The `set_printoptions()` function has multiple options, but if only a single integer is given, it controls the maximum number of digits that

can appear after the decimal point.

```
#!/usr/local/bin/python3

import numpy as np
import sys

x = np.asarray([1, 0, 0], np.int32)
y = np.asarray([ [1, 2, 3.2454], [4,5,.0468], [7,8,9.1111] ], np.float64)

print("x = ", x, "\ny = \n", y)

np.set_printoptions(2)
print("x = ", x, "\ny = \n", y)
```

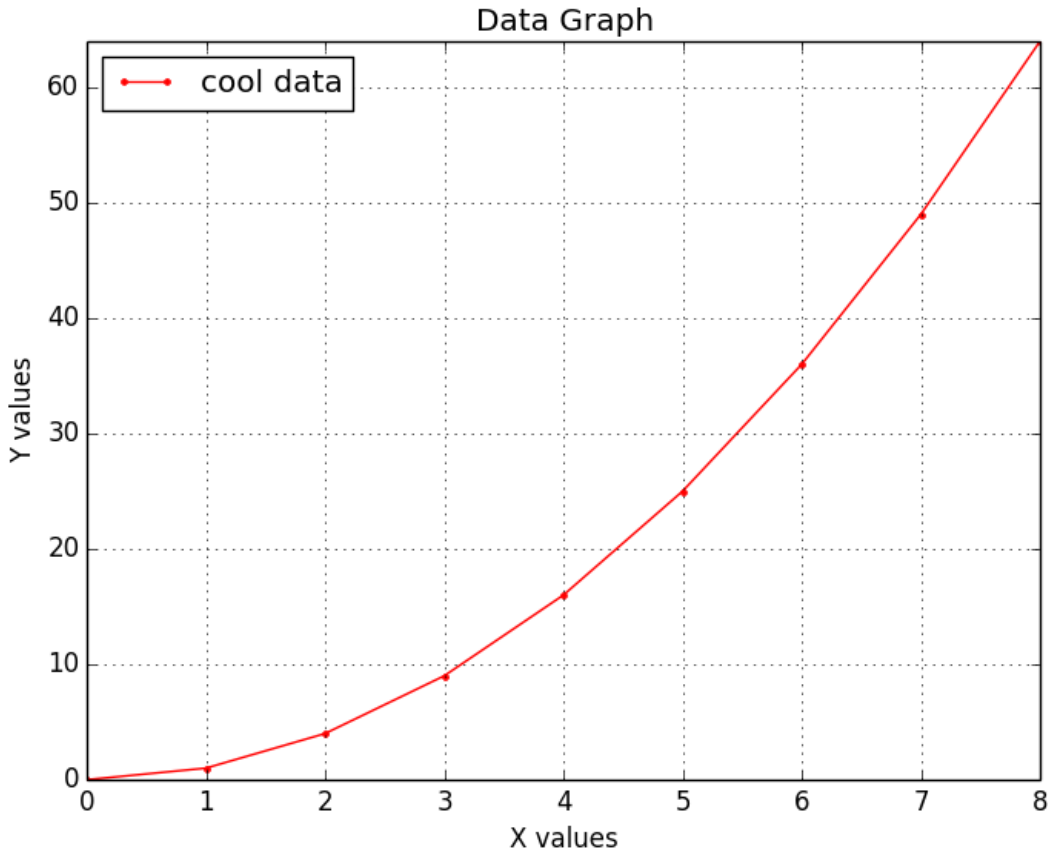
9 Plotting

Generating graphs in Python is easy using matplotlib, and the following example assumes that package has been installed.

```
#!/usr/local/bin/python3
import sys
import matplotlib.pyplot as plt

if( len(sys.argv) != 2 ) :
        print( "usage is: %s datafile.txt" % (sys.argv[0]) )
        sys.exit()

fd = open( sys.argv[1], 'r' )
x = []
y = []
for line in fd :
  xtext, ytext = line.split()
  x.append( float(xtext) )
  y.append( float(ytext) )
minx = min(x)
maxx = max(x)
miny = min(y)
maxy = max(y)
```

```
plt.title(  'Data Graph' )
plt.xlabel( 'X values' )
plt.ylabel( 'Y values' )
plt.xticks()
plt.yticks()
plt.grid('on')
plt.plot(x, y, '-r.', label='cool data', linewidth=1)
# Can plot another line on the same graph like this
# plt.plot(w, z, '--g<', label='hot data', linewidth=2)
plt.xlim(minx, maxx)
plt.ylim(miny, maxy)
plt.legend(loc='best')
plt.ion()
plt.show()

fd.close()
value = input('Hit CR to continue\n')
```

The input to this program is a text file with 2 entries per line: an x value and a y value. The x and y values are read one line at a time and separate lists of floats are used to hold them. The maximum and minimum values in the x and y lists are then found and used later to establish the limits on the x and y axes. The calls to the plotting module are reasonably self-explanatory. Using it, a title is set as are x-axis and y-axis labels. Tic marks and grid lines are also enabled. Note that the call to `plot( )` uses the format string `'-r.'` which specifies that the line should be solid, red, and the data points should be marked with a '.' The call to `ion( )` turns on 'interactive mode' which allows the user to continue interacting with the program via the keyboard. Finally, the call to `draw( )` causes the graph to appear.



The figures below show how to use different line styles, colors, and marker styles. For example `'- -g<'` specifies a green dashed line with triangle-left markers. Two or more data lines can be drawn on the same graph with multiple calls to `plot( )`.

The following format string characters are accepted to control the line style or marker:

character	description
`'-'`	solid line style
`'--'`	dashed line style
`'-.'`	dash-dot line style
`':'`	dotted line style
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle_down marker
`'^'`	triangle_up marker
`'<'`	triangle_left marker
`'>'`	triangle_right marker
`'1'`	tri_down marker
`'2'`	tri_up marker
`'3'`	tri_left marker
`'4'`	tri_right marker
`'s'`	square marker
`'p'`	pentagon marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'D'`	diamond marker
`'d'`	thin_diamond marker
`'|'`	vline marker
`'_'`	hline marker

The following color abbreviations are supported:

character	color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

10 Math

Python has several built-in math functions that can be accessed without including any modules. These include absolute value and exponentiation: `abs()` and `pow( )` (note that ** can also be used for exponention). A rounding function for floats is also available; for example, `round(`x`, 3)` rounds x to 3 digits (there is a numpy version of round which works on numpy arrays). There are also useful conversion functions like `hex( )` that convert an integer into a hexadecimal string, and `oct( )` which does the same for octal. The 'cast' functions `int( ), float( )`, and `complex( )`

are also built-in.

10.1 Standard "C" math functions

In addition to the built-in functions, Python has an always-available module named `math` that provides access to standard "C" math functions. Functions such as `math.ceil( ), math.floor( ), math.fabs( ), math.exp( ), math.log( ), math.pow( )`, and `math.sqrt( )` are therefore easily accessed by including `import math` at the top of your script. Of course, the standard trigonometric functions are also available via `math.sin( ), math,cos( ), math.tan( ), math.asin( ), math.acos( ), math.atan( )`, and `math.atan2( )`.

Important constants are also included, for example, `math.pi` will give the value of π and `math.e` the value of e. Hyperbolic functions and some other special functions such as `math.erf( )` are also supported.

When working with numpy, you will find times when you want to apply a math function to every element of a numpy array. Generally speaking, there is always an equivalent numpy function available. For example, `np.sin( x )` will take the `sin` of every element of the numpy array `x`.

In case you have ever worked with Python 2, you should note that in python 3, the division of 2 integers returns a floating point number, not the integer part. To obtain just the integer part when dividing integers, use `//`. For example 8//3 is equal to 2. The remainder of the division of two integers is obtained using `%` for example, 8%5 is equal to 3.

10.2 Complex numbers

Python has an always-available module named `cmath` that provides math functions for complex numbers. The functions in this module accept integers, floats, and complex numbers as arguments. Complex numbers are stored internally in rectangular form with a real and an imaginary part. Note that the built-in Python functions `abs( )` and `pow( )` will also accept complex arguments. The following script illustrates the use of several `math` and `cmath` functions.

```
#!/usr/local/bin/python3
import math
import cmath
```

```
# Initialize by specifying the real and imaginary parts
x = complex(1.0, 2.0)
y = 4 * x
print( "real part = ", y.real, "imaginary part = ", y.imag, "y = ", y )

# Can also initialize like this
z = 1+2j
print( "z = ", z, " z conjugate = ", z.conjugate() )

# Convert rectangular to polar coordinates
m, p = cmath.polar(z)

#Or can use two calls to get mag and phase separately
#m = abs(z)
#p = cmath.phase(z)
print( "magnitude = ", m, " phase = ", p )

# Convert polar to rectangular coordinates
z = cmath.rect(m, p)
print( "z = ", z )

# Some other operations for illustration:
# raising  e to a complex number:
w = pow( math.e, complex(0, math.pi/4) )
print( "w = ", w)
# taking the log of a complex number
b = cmath.log(w)
print( "b = ", b )
```

10.3 Element-by-element Array math with numpy

Numpy allows many operations to be performed on arrays using a compact notation reminiscent of Matlab. The following script demonstrates some of numpy's power. By googling you can find many additional examples.

```
#!/usr/local/bin/python3
import numpy as np

# Initialize array x of size 3 to 0, 1, 2
```

```
x = np.asarray( list(range(3)), np.float)

# Add 5 to each element of x
y = x + 5

print("x = \n", x, "\ny = \n", y)

# element-by-element operations are supported, such as *,
# +, /, etc.  For example
print( x+y, x*y, y**x )

# element-by-element operations with scalars are
# supported too.  We used this above with +.  Examples:
print("x = \n", x )
print( x-2, x/2 )

# Built-in functions can be applied to arrays
# Here we use ":" to indicate every entry in array w[]
w = np.zeros(3, np.complex128)
w[:] = 1 +1j
print( w, abs(w), w**x )

# Other operations on complex arrays can be done as follows
print( w.real, w.imag, np.conj(w) )

# Some other useful array functions include
print("\n", np.sum(x), np.log(y) )
```

One thing worth highlighting is the ease with which the values in an array can be conditionally modified. For example, if d is a numpy array, then every location in d with a value greater than 15 can be easily modified (in this example set to -1) as shown below

```
#!/usr/local/bin/python3
import numpy as np

x = np.asarray( list(range(20)), np.float)
print("x = \n", x)
x[ x > 15 ] = -1
print("x = \n", x)
```

10.4 The FFT and statistics

An important transform in many fields of engineering is the Discrete Fourier Transform. The DFT is well supported by numpy as the following example shows. This script also demonstrates several functions that facilitate statistical calculations.

```
#!/usr/local/bin/python3
import numpy as np

# 1-D DFT calculations
x = np.zeros(4, np.complex128)
for i in range(4):
  x[i] = complex(2*i, i+3)
y = np.fft.fft(x)
print( "x = ", x )
print( "DFT(x) = ", y )
print( "Magnitude of DFT(x) = ", abs(y) )
xr = np.fft.ifft(y)
print( "inverse DFT result = x = ", xr )

# 2-D DFT calculations
w = np.zeros( (4,4), np.complex128 )
for i in range(4) :
  for j in range(4) :
    w[i, j] = complex(i+3*j, i*j)
print( "w = \n", w )
dft_w = np.fft.fft2(w)
print( "dft_w = \n", dft_w )
idft = np.fft.ifft2(dft_w)
print( "idft = w = \n", idft )

# Stats calculations.  Note how we convert a
# list to a float64 array in this example, and
# compute the for loop for values between
# 13 and 400 incrementing in steps of 3
l = []
for i in range(13, 400, 3) :
  l.append(i)
a = np.asarray(l, np.float64)
print( "min(a) = ", np.amin(a), "max(a) = ", np.amax(a) )
print( "sum(a) = ", np.sum(a) )
```

```
print( "mean(a) = ", np.mean(a), "median(a) = ", np.median(a) )
print( "var(a) = ", np.var(a), "std(a) = ", np.std(a) )
```

The cumulative normal distribution can be computed using the math module's `erf( )` function (the error function) as follows:

```
#!/usr/local/bin/python3
import math

def cumnorm(mean, var, x) :
  tmp = (x - mean) / math.sqrt(var)
  return 0.5 + 0.5 * math.erf( tmp / math.sqrt(2) )

print( cumnorm(0, 1, -2)   )
print( cumnorm(0, 1, -1)   )
print( cumnorm(0, 1,  0)   )
print( cumnorm(0, 1,  1)   )
print( cumnorm(0, 1,  2)   )
```

10.5 Linear algebra with numpy

Below is a script illustrating how many common linear algebra operations on matrices and vectors can be performed.

Dealing with 1D vectors can be tricky in numpy. In order to make sure that 1D vectors have an orientation, i.e., are treated as either row vectors or column vectors, they cannot simply be created as 1D numpy arrays. They must be created as 2D arrays of size `(N,1)` for a column vector, or of size `(1,N)` for a row vector. It is possible to multiply 1D arrays and 2D arrays together, however, no orientation information is checked in this case. In other words, a 1D numpy array can left multiply a 2D numpy array (it is safe to think of a 2D array as a matrix) or it can be left-multiplied by a 2D array. In one case numpy treats the 1D array as a row vector, and in the other case as a column vector. Furthermore, if `v` is a 1D array, the transpose operation applied to `v` *has no effect*.

In order to more closely model the intended mathematics, I highly recommend creating *vectors* as 2D numpy arrays with either one row or one column, and viewing them as *distinct from 1D arrays*. A 1D numpy array of size N can be easily turned into a column vector as follows:

```
v_col_vec = np.reshape(v, (N,1))
```

Or it can be turned into a row vector like this:

```
v_row_vec = np.reshape(v, (1,N))
```

Finally, here are more compact ways to create row and column vectors without using `np.reshape()`:

```
v_row_vec = np.asarray([v])

v_col_vec = np.asarray([v]).T
```

The following script demonstrates more linear algebra manipulations.

```
#!/usr/local/bin/python3
import numpy as np
from numpy import linalg as LA

# Create 3 1-D column vectors of size 32 integers
tmp = np.reshape( np.asarray( list( range(3) ), np.int32 ), (3,1) )
c1 =  tmp + 1
c2 =  9 - tmp
c3 =  5 + 2*(tmp+1) + tmp**2
print(f"c1= \n{c1}\n c2= \n{c2}\n c3=\n{c3}\n")

# Take the transpose to convert a column vector
# into a row vector
c1T = c1.T
print(f"c1.T ={c1T}\n")

# Stack the vectors horizontally to create a matrix
# The first vector becomes the first column, etc.
c_horiz = np.hstack( (c1, c2, c3) )
print(f"horizontal stacking =\n{c_horiz}\n")

# Stack the vectors vertically to create a matrix
# Note that we take the transpose so that the first
# vector becomes the first row, etc.
c_vert = np.vstack( (c1.T, c2.T, c3.T) )
```

```
print(f"vertical stacking =\n{c_vert}\n")

# You can also stack Matrices horizontally and
# vertically.  Look at these outputs
Xh = np.hstack( (c_horiz, c_vert) )
print(f"Xh =\n{Xh}\n")
Xv = np.vstack( (c_horiz, c_vert) )
print(f"Xv =\n{Xv}\n")

# Sometimes you want to add a new dimension.  For
# example, you might have R, G, and B matrices
# and want to create a new matrix with R in plane
# 0, G in plane 1, and B in plane 2.  This can
# be done using np.stack()
Xsf = np.stack( (c_horiz, c_vert) )
print(f"Xsf=\n{Xsf}\n")

# If you want the new dimension to be added
# at the end, you can do this instead
Xsl = np.stack( (c_horiz, c_vert), axis=-1 )
print(f"Xsl=\n{Xsl}\n")

# you can get the shape of a matrix like this
print(f"Xsl's shape= {Xsl.shape}\n")

# Xsl.size gives the product of the array dimensions
print("Xsl.size = ", Xsl.size )

# Create a 1-D array from a 2-D matrix
Q = c_horiz.flatten() # goes in row order
print("c_horiz after row-major flattening", Q)
Q = c_horiz.flatten('F') # goes in column order
print("c_horiz after column-major flattening", Q)

# Cast the array type to float using "astype"
X = c_horiz.astype(np.float64)

# Element-by-element matrix operations
print(f"X =\n{X}\n")
Y = X + 1 ; print( "Y = \n", Y )
Z = X * Y ; print( "Z = \n", Z )
```

```
# Extract one row or one column
# note the use of slicing
print( "first row = ", Z[0] )
print( "third column = ", Z[:,2] )

# Matrix multiply. Use V = np.dot(X, Y) or
# V = X.dot(Y) or this compact form:
V = X@Y
print( "XY = \n", V )

# Matrix transpose and determinant
print( "X transpose = \n", X.T )
print( "det(X) = ", LA.det(X)  )

# Matrix eigenvalues. Catch errors
try:
  w, V = LA.eig(X)
except: print( "eigenvector calculation failed" )
else:
  print( "eigenvalues = ", w )
  print( "eigenvectors are the columns = \n", V )

# matrix inverse: catch errors
try:
  W = LA.inv(X)
except: print( "Exception raised: not invertible" )
else :
  print( "X = \n", X )
  print( "W = Xinverse = \n", W )
  print( "XW = Identity = \n", np.round(np.dot(X, W), 2) )

# Mixed matrix and vector operations
v = np.asarray( [[1., 2., 3.]] ).T
print( "X = \n", X )
print( "v = ", v )
print( "Xv = ", X@v ) #left-multiply v by X

#transpose v then right-multiply X by v.T
print( "v.T X = ", v.T@X )
print( "inner vector product = ", v.T@v )
```

```
print( "outer vector product = \n", v@v.T )
print( "magnitude of vector = ", LA.norm(v) )

# Solve a matrix equation 2 ways.
# c is solution to Xc = v
print( "Solve Xc = v for c:" )
np.set_printoptions(2)
print( "W = Xinverse = \n", np.round(W,2), "\nv =\n", v, "\n" )
print( "c = W@v =\n", np.round(W@v, 2), "\n" )
print( "Solve Xc = v for c using method 2:" )
c = LA.solve(X, v)
print( "c =\n", c, "\nXc =\n", X@c )
```

To reiterate the comment made in the code, the `np.stack()` command is interesting when wanting to combine multiple arrays together while adding a dimension to the combination. The new dimension can most obviously be added as either the first index of the new array, or the last. Each of the arrays in the list passed to `np.stack()` can then be accessed by specifying the array you want by indexing the new dimension. This can be useful when dealing with arrays of arbitrary size. For example, 3D image arrays of shape `(rows, cols, color_planes)` can be stacked into a 4-D array of 3D images.

10.6 Random numbers and the SVD

The following script demonstrates a few capabilities from the 'random' module.

```
#!/usr/local/bin/python3
import random

# Can give seed() an integer argument to get the same
# random sequence twice.  Otherwise, using no
# argument will initialize the generator from
# an internal OS clock
random.seed()

# Get some random ints satisfying 10 <= n <= 100
for i in range(10) :
  print( "random int %d = %d" % (i, random.randint(10, 100)) )

# Generate a list of ints from 0 to 20
```

```
# and randomly shuffle them in place
x = list(  range(0, 20, 1)  )
random.shuffle(x)
print( x )

# Get 10 uniformly distributed random floats
# satisfying 1 <= x <= 3.  Note the compact list
# initialization syntax
x = [random.uniform(1, 3) for i in range(10)]
print( x )

# Get 10 normally distributed random floats
# with mean=1 and std=3
x = [random.gauss(1, 3) for i in range(10)]
print( x )
```

The next script is the most conceptually complicated example in this document. It demonstrates how to perform a Singular Value Decomposition (SVD). Recall that, given a set of multi-dimensional measurements, the SVD finds the best set of basis vectors in the squared-error sense for approximating the measurements; these basis vectors are the principal components. In other words, the basis vectors are chosen to minimize the residual squared error when the measurement data is projected onto the sub-space spanned by a subset of the principal components. For example, for a 10-D measurement set, 10 principal components will be found, because 10 vectors span the space. The first principal component—among all possible unit vectors that could be chosen—will be the best for minimizing the total residual squared error when the measurement data is projected onto just one vector. The first two principal components—among all possible pairs of perpendicular unit vectors that could be chosen—will be the best for minimizing the total residual squared error when the measurement data is projected onto just two vectors. The SVD allows the dimensionality of a data set to be reduced while minimizing the error power.

```
#!/usr/local/bin/python3
import random
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA

# Declare a matrix to hold 2-D measurement data.  Also
# initialize two row vectors orthogonal to each other
```

```
M = np.zeros( (500, 2), np.float64 )
u0 = np.asarray( [[3., 2.]], np.float64 )
u1 = np.asarray( [[-2., 3.]], np.float64 )

# Normalize u0 and u1: make them unit vectors
u0 = u0 / LA.norm(u0)
u1 = u1 / LA.norm(u1)

# Generate some random data clustered around these axes
for i in range(500) :
  M[i] = random.uniform(0,10)*u0 + random.gauss(0,1)*u1

plt.title('Measurement Data')
plt.xticks()
plt.yticks()
plt.grid('on')
plt.plot(M[:,0], M[:,1], 'r.', label='raw data', linewidth=1)
plt.ion()
plt.show()

# Do the SVD on M
U, s, VT = LA.svd(M)

# print the principal components: these are the
# rows of VT (i.e the cols V=VT.T)
print( f"u0 = {np.round(u0,3)}")
print( f"component 0 = {np.round(VT[0], 3)}")
print( f"u1 = {np.round(u1,3)}")
print( f"component 1 = {np.round(VT[1], 3)}")
value = input('Hit CR to continue\n')
```

11 Other Useful Operations

11.1 Miscellaneous

11.1.1 np.where()

When dealing with an array, frequently the need arises to perform an operation on its elements based on a logical operation performed on its elements. As an example, you might want to set every value of an array equal to 1 if it is greater than or equal to 0, and to -1 if it is less than zero (an operation called hard limiting). The numpy function `np.where()` is very useful in this and similar cases. The typical usage looks like this:

```
D = np.where(logical condition on array 'A', B, C)
```

Here a logical operation is performed on each element of array A, and if it evaluates to `True`, it is replaced by the corresponding element of array `B`, otherwise with the corresponding element of array `C`. For the hard limiting case described above the code would look like this:

```
D = np.where(A >= 0, 1, -1)
```

11.1.2 Enumerating while looping over list elements

Python `for` loops step through each element of a list. The elements of the list need not be numbers. Often when stepping through a list you need to know an element's index. This can be easily obtained using `enumerate` as shown in the two examples below:

```
#!/usr/local/bin/python3
import numpy as np

# Create a list to loop over
mydata = np.asarray( [2*i for i in range(10)] )
for cnt, datval in enumerate(mydata):
  print(f"cnt = {cnt} datval = {datval}")

# Another example
names = ("Jack", "Jill", "Sam", "Cheri")
for cnt, name in enumerate(names):
  print(f"cnt = {cnt} name = {name}")
```

11.1.3 Dynamically growing a 2D numpy array row-by-row

This example shows how to grow a 2D numpy array, whose number of columns is initially unknown, one row at a time:

```
#!/usr/local/bin/python3
import sys
import numpy as np

# Create an empty array
inputs = np.array( [] )

# Now loop and add data
for i in range(7):
  tmp = np.asarray( [i, 2*i, 3*i] )
  inputs  = np.vstack( [inputs,  tmp] ) if inputs.size else tmp

print(f"inputs = \n{inputs}")
```

11.1.4 Executing OS commands and shell functions

Sometimes it is useful to execute an operating system or a shell command from within a script. This is supported by the os module as shown below:

```
#!/usr/local/bin/python3
import os

# Unix shell example.  Put some text in a file,
# dump the file to the screen, then delete it

cmd_line = "echo how are you &> zzz_foo.txt"
os.system(cmd_line)
cmd_line = "cat zzz_foo.txt; rm -f zzz_foo.txt"
os.system(cmd_line)
```

11.2 Python Classes

The script below demonstrates the basic use of classes (minus inheritance which we will not need).

```
#!/usr/local/bin/python3
import sys
import numpy as np

class myCoolWidget(object) :
  # Variables defined at the class level and not inside a class method
  # are static.  There is only ONE such variable per class.  Static variable
  # are accessed via myCoolWidget.varname and NOT with an instance name.
  # In other words, for the variables below, access them with
  # myCoolWidget.widget_catetory and myCoolWidget.widget_brand

  widget_category = 1
  widget_brand = 3.14

  # Below is a static variable that is NOT visible outside the class
  # because it starts with __.  It can only be used inside class methods
  # It too is accessed with myCoolWidget.varname (in this case
  # myCoolWidget.__sf)

  __sf = 2

  # NOTE:
  # Variables declared INSIDE a class method that are preceded with
  # "self." are instance variables.  There is one variable per class
  # instance.  Variables preceded by __ are private to the class
  # and NOT visible outside the class.  Variables not preceded by __ can
  # be accessed outside the class using a class instance.
  # For example, if "A" is a class instance,
  # the variable uCanSeeMe defined below can be accesssed
  # as A.uCanSeeMe.  However,
  # the variable __xD cannot be accessed via A.__xD since it is private.
  # The "self" word in class methods can be thought of as creating an
  # instance variable.  Without the "self." a variable is local in scope
  # to its method only.

  # Constructor. Assume the first 3 arguments are scalars and
  # the last argument is a vector.  OBSERVE: the vector is copied so that
  # the instance can change the vector without changing the array
  # passed into the constructor (arrays are mutable!).  Scalars are
  # immutable, so an assignment with "=" achieves a "copy".
```

```
  def __init__(self, xDimension, yDimension, zDimension, performanceVals ) :
    # Make private copies for later use by class methods only.
These are
    # not visible outside the class
    self.__xD = xDimension # width
    self.__yD = yDimension # depth
    self.__zD = zDimension # height
    self.__pV = np.copy( performanceVals ) # vector of miscellaneous values

    # Compute some values we want
    self.__volume = self.__xD * self.__yD * self.__zD
    self.__dimSum = self.__xD + self.__yD + self.__zD

    # Create a variable that is visible outside the class with an instance
    # reference: a "public" variable.  The "self" is required, otherwise
    # the variable just has local scope to this method
    self.uCanSeeMe = 123.456

  # Here are three simple class methods.
  # The "self" is mandatory syntax (look at the __init__ syntax
  # above to see how to pass multiple variables if needed)

  def get_widget_volume(self) :
    return self.__volume

  def get_widget_dimSum(self) :
    return self.__dimSum

  def echo_perf_values(self) :
    print("Scaled performance values are:\n", myCoolWidget.__sf * self.__pV)

##############
#MAIN PROGRAM
##############

# Create two instances
stereo = myCoolWidget(8, 8, 2, np.asarray([22.3, 34.5, 66.7], np.float32) )
television = myCoolWidget(50, 2, 50, np.asarray([12.3, 13.5], np.float32) )

# Access and change a class variable
print("myCoolWidget.widget_category = ", myCoolWidget.widget_category)
```

```
myCoolWidget.widget_category = 2
print("myCoolWidget.widget_category = ", myCoolWidget.widget_category)

# Demonstrate public instance variables
stereo.uCanSeeMe = 23
television.uCanSeeMe = 45
print("stereo.uCanSeeMe = ", stereo.uCanSeeMe)
print("television.uCanSeeMe = ", television.uCanSeeMe)

# The lines below would FAIL since these are private variables
# print("television.__xD = ", television.__xD)
# print("myCoolWidget.__sf = ", myCoolWidget.__sf)

# Now call some class methods
print("stereo.get_widget_volume() = ", stereo.get_widget_volume() )
print("stereo.get_widget_dimSum() = ", stereo.get_widget_dimSum() )
stereo.echo_perf_values()

print("television.get_widget_volume() = ", television.get_widget_volume() )
print("television.get_widget_dimSum() = ", television.get_widget_dimSum() )
television.echo_perf_values()
```

11.3 Reading/Writing an Excel Comma Separated Values (CSV) file

It is often necessary to process CSV formatted data, especially in Excel's format. Many devices and programs produce such data for data interchange purposes. Python provides support for CSV data via the csv module. The following script demonstrates the module's use for Excel's CSV flavor.

```
#!/usr/local/bin/python3
import csv
import sys

if len(sys.argv) != 3 :
  print( "usage is: %s input.csv output.csv" % (sys.argv[0]) )
  sys.exit()

# Open files and create csv readers and writers
```

```
fd_r = open(sys.argv[1], 'r')
fd_w = open(sys.argv[2], 'w')
datareader = csv.reader(fd_r, dialect='excel')
datawriter = csv.writer(fd_w, dialect='excel')

# Data has 3 columns, each column has a label
# (Name, Age, and Birthdate).  Read the labels
# then the data in the file
label1, label2, label3 = next(datareader)
print( label1, label2, label3 )

for val in datareader :
  name = val[0]
  age =  val[1]
  bday = val[2]
  print( name, age, bday )

# Now write a csv file.  First the column labels,
# then some data
row = ['Grocery store item', 'Quantity', 'Price']
datawriter.writerow(row)
for i in range(10) :
  row[0] = "orange_" + str(i)
  row[1] = str( 10*i )
  row[2] = str( 12.23 + i )
  datawriter.writerow(row)

fd_r.close()
fd_w.close()
```

11.4 Using lists

Lists are one of Python's most useful objects, and there are many functions to facilitate their manipulation. One important thing to know about them is that the same list can hold different data types; lists can even be elements of another list. The following script shows some common list operations.

```
#!/usr/local/bin/python3
```

```
# Create a list of strings and one of floats
a = ["adam", "jane", "bill", "kelly", "charlie"]
b = [10.0*i for i in range(5)]

# Make a list of these two lists
c = [a, b]
print( "c[0] = ", c[0] )
print( "c[1] = ", c[1] )
print( "c[0][2] = ", c[0][2] )
print( "c[1][2] = ", c[1][2] )

# Check if an element is in list c[0];
# If it is, increment the corresponding
# float in the list c[1]
if "bill" in c[0] :
  i = c[0].index("bill")
  c[1][i] += 1
print( "c[1] = ", c[1] )

# access a subset of a list: note that the
# last index is NOT used, just like range()
print( "b[1] to b[3] = ", b[1:4] )

# add values to both lists stored in c
c[0].append('cathy')
c[1].append(50.0)
print( "c = \n", c )

# delete a value from the list c[0]
del c[0][2]
print( "c = \n", c )

# Create a new list and see how often
# the values `4' and jane occur
d = [25, 4, "jack", 58, "jane", 4, 1, 28, 4, "jane"]
print( d )
print( d.count(4), d.count('jane') )

# Sort the list a
f = sorted( a )
print( "a sorted = ", f )
```

```
# Combine lists a and d into one list
# Note that `bill' was deleted and `cathy' was
# added.  These changes made using variable 'c'
# affected the values seen by variable 'a'
e = a + d
print( "e = \n", e )
```

12 Supporting Information

The following websites have useful introductory material

1. `https://docs.python.org/3.1/tutorial/introduction.html`
2. `https://www.tutorialspoint.com/python3/python_numbers.htm`
3. `https://realpython.com/`

The following figures provide a fuller list of the operations available for numpy arrays.

Trigonometric functions

`sin` (x, /[, out, where, casting, order, ...])	Trigonometric sine, element-wise.
`cos` (x, /[, out, where, casting, order, ...])	Cosine element-wise.
`tan` (x, /[, out, where, casting, order, ...])	Compute tangent element-wise.
`arcsin` (x, /[, out, where, casting, order, ...])	Inverse sine, element-wise.
`arccos` (x, /[, out, where, casting, order, ...])	Trigonometric inverse cosine, element-wise.
`arctan` (x, /[, out, where, casting, order, ...])	Trigonometric inverse tangent, element-wise.
`hypot` (x1, x2, /[, out, where, casting, ...])	Given the "legs" of a right triangle, return its hypotenuse.
`arctan2` (x1, x2, /[, out, where, casting, ...])	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
`degrees` (x, /[, out, where, casting, order, ...])	Convert angles from radians to degrees.
`radians` (x, /[, out, where, casting, order, ...])	Convert angles from degrees to radians.
`unwrap` (p[, discont, axis])	Unwrap by changing deltas between values to 2*pi complement.
`deg2rad` (x, /[, out, where, casting, order, ...])	Convert angles from degrees to radians.
`rad2deg` (x, /[, out, where, casting, order, ...])	Convert angles from radians to degrees.

Hyperbolic functions¶

`sinh` (x, /[, out, where, casting, order, ...])	Hyperbolic sine, element-wise.
`cosh` (x, /[, out, where, casting, order, ...])	Hyperbolic cosine, element-wise.
`tanh` (x, /[, out, where, casting, order, ...])	Compute hyperbolic tangent element-wise.
`arcsinh` (x, /[, out, where, casting, order, ...])	Inverse hyperbolic sine element-wise.
`arccosh` (x, /[, out, where, casting, order, ...])	Inverse hyperbolic cosine, element-wise.
`arctanh` (x, /[, out, where, casting, order, ...])	Inverse hyperbolic tangent element-wise.

Rounding

`around` (a[, decimals, out])	Evenly round to the given number of decimals.
`round_` (a[, decimals, out])	Round an array to the given number of decimals.
`rint` (x, /[, out, where, casting, order, ...])	Round elements of the array to the nearest integer.
`fix` (x[, out])	Round to nearest integer towards zero.
`floor` (x, /[, out, where, casting, order, ...])	Return the floor of the input, element-wise.
`ceil` (x, /[, out, where, casting, order, ...])	Return the ceiling of the input, element-wise.
`trunc` (x, /[, out, where, casting, order, ...])	Return the truncated value of the input, element-wise.

Sums, products, differences

`prod` (a[, axis, dtype, out, keepdims])	Return the product of array elements over a given axis.
`sum` (a[, axis, dtype, out, keepdims])	Sum of array elements over a given axis.
`nanprod` (a[, axis, dtype, out, keepdims])	Return the product of array elements over a given axis treating Not a Numbers (NaNs) as ones.
`nansum` (a[, axis, dtype, out, keepdims])	Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.
`cumprod` (a[, axis, dtype, out])	Return the cumulative product of elements along a given axis.
`cumsum` (a[, axis, dtype, out])	Return the cumulative sum of the elements along a given axis.
`nancumprod` (a[, axis, dtype, out])	Return the cumulative product of array elements over a given axis treating Not a Numbers (NaNs) as one.
`nancumsum` (a[, axis, dtype, out])	Return the cumulative sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.
`diff` (a[, n, axis])	Calculate the n-th discrete difference along given axis.
`ediff1d` (ary[, to_end, to_begin])	The differences between consecutive elements of an array.
`gradient` (f, *varargs, **kwargs)	Return the gradient of an N-dimensional array.
`cross` (a, b[, axisa, axisb, axisc, axis])	Return the cross product of two (arrays of) vectors.
`trapz` (y[, x, dx, axis])	Integrate along the given axis using the composite trapezoidal rule.

Exponents and logarithms

`exp` (x, /[, out, where, casting, order, ...])	Calculate the exponential of all elements in the input array.
`expm1` (x, /[, out, where, casting, order, ...])	Calculate `exp(x) - 1` for all elements in the array.
`exp2` (x, /[, out, where, casting, order, ...])	Calculate *2**p* for all *p* in the input array.
`log` (x, /[, out, where, casting, order, ...])	Natural logarithm, element-wise.
`log10` (x, /[, out, where, casting, order, ...])	Return the base 10 logarithm of the input array, element-wise.
`log2` (x, /[, out, where, casting, order, ...])	Base-2 logarithm of *x*.
`log1p` (x, /[, out, where, casting, order, ...])	Return the natural logarithm of one plus the input array, element-wise.
`logaddexp` (x1, x2, /[, out, where, casting, ...])	Logarithm of the sum of exponentiations of the inputs.
`logaddexp2` (x1, x2, /[, out, where, casting, ...])	Logarithm of the sum of exponentiations of the inputs in base-2.

Arithmetic operations

`add` (x1, x2, /[, out, where, casting, order, ...])	Add arguments element-wise.
`reciprocal` (x, /[, out, where, casting, ...])	Return the reciprocal of the argument, element-wise.
`negative` (x, /[, out, where, casting, order, ...])	Numerical negative, element-wise.
`multiply` (x1, x2, /[, out, where, casting, ...])	Multiply arguments element-wise.
`divide` (x1, x2, /[, out, where, casting, ...])	Divide arguments element-wise.
`power` (x1, x2, /[, out, where, casting, ...])	First array elements raised to powers from second array, element-wise.
`subtract` (x1, x2, /[, out, where, casting, ...])	Subtract arguments, element-wise.
`true_divide` (x1, x2, /[, out, where, ...])	Returns a true division of the inputs, element-wise.
`floor_divide` (x1, x2, /[, out, where, ...])	Return the largest integer smaller or equal to the division of the inputs.
`float_power` (x1, x2, /[, out, where, ...])	First array elements raised to powers from second array, element-wise.
`fmod` (x1, x2, /[, out, where, casting, ...])	Return the element-wise remainder of division.
`mod` (x1, x2, /[, out, where, casting, order, ...])	Return element-wise remainder of division.
`modf` (x[, out1, out2], / [[, out, where, ...])	Return the fractional and integral parts of an array, element-wise.
`remainder` (x1, x2, /[, out, where, casting, ...])	Return element-wise remainder of division.
`divmod` (x1, x2[, out1, out2], / [[, out, ...])	Return element-wise quotient and remainder simultaneously.

Handling complex numbers

`angle` (z[, deg])	Return the angle of the complex argument.
`real` (val)	Return the real part of the complex argument.
`imag` (val)	Return the imaginary part of the complex argument.
`conj` (x, /[, out, where, casting, order, ...])	Return the complex conjugate, element-wise.

Miscellaneous

`convolve` (a, v[, mode])	Returns the discrete, linear convolution of two one-dimensional sequences.
`clip` (a, a_min, a_max[, out])	Clip (limit) the values in an array.
`sqrt` (x, /[, out, where, casting, order, ...])	Return the positive square-root of an array, element-wise.
`cbrt` (x, /[, out, where, casting, order, ...])	Return the cube-root of an array, element-wise.
`square` (x, /[, out, where, casting, order, ...])	Return the element-wise square of the input.
`absolute` (x, /[, out, where, casting, order, ...])	Calculate the absolute value element-wise.
`fabs` (x, /[, out, where, casting, order, ...])	Compute the absolute values element-wise.
`sign` (x, /[, out, where, casting, order, ...])	Returns an element-wise indication of the sign of a number.
`heaviside` (x1, x2, /[, out, where, casting, ...])	Compute the Heaviside step function.
`maximum` (x1, x2, /[, out, where, casting, ...])	Element-wise maximum of array elements.
`minimum` (x1, x2, /[, out, where, casting, ...])	Element-wise minimum of array elements.
`fmax` (x1, x2, /[, out, where, casting, ...])	Element-wise maximum of array elements.
`fmin` (x1, x2, /[, out, where, casting, ...])	Element-wise minimum of array elements.
`nan_to_num` (x[, copy])	Replace nan with zero and inf with finite numbers.
`real_if_close` (a[, tol])	If complex input returns a real array if complex parts are close to zero.
`interp` (x, xp, fp[, left, right, period])	One-dimensional linear interpolation.