Python3 for Engineers

prepared for MECH3750 "Engineering Analysis II" at
The University of Queensland
by Tom Reddell

Contents

1 Libraries and Documentation

1.1 Important Libraries

A library is a collection of algorithms or data structures which build upon the basic python built in capabilities. There are over 100 000 python libraries available for download from the Python Package Index (PyPI) [1], some of the most important libraries for engineering and scientific analysis are listed below:

Table 1: Important Libraries for Scientific and Engineering Analysis

Library	**Description**
math	Part of the python standard library, contains many inbuilt math functions and constants.
numpy	A fundamental package for scientific computing, numpy adds multi-dimensional arrays and linear algebra routines, as well as Fourier transforms.
scipy	A large collection of algorithms for scientific computing, contains optimisation, linear algebra, integration, interpolation, fast Fourier transform, signal and image processing, as well as special mathematical functions.
matplotlib	2 and 3 dimensional plotting routines and visualisation tools.

1.2 Inbuilt Help

You can view the contents of a library or module by using the `dir` command. For example, to view the contents of the math library:

```
>> import math
>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'tau', 'trunc']
```

To get documentation on a specific function or module, we can run the `help` function. For example, to understand the functionality of the `atan2` function:

```
>> help(math.atan2)
```

```
Help on built-in function atan2 in module math:

atan2(...)
    atan2(y, x)

    Return the arc tangent (measured in radians) of y/x.
    Unlike atan(y/x), the signs of both x and y are considered.
```

1.3 Online Help

The python help page (https://www.python.org/about/help/) contains many links to documentation, tutorials and additional official and non official support options.

[1] https://pypi.python.org/pypi

2 Python Fundamentals

2.1 Importing Modules

In order to import a module, it must be located in the python path, by default, this will be your local python install directory, as well as your current working directory. There are several different ways to import a module.

- `import <package>` will load the package and its entire contents as a namespace. You will be able to access its contents using dot notation. For example:

  ```
  >>import math
  >>math.pi
  3.141592653589793
  ```

- `import <package> as <reference>`. The same as above, except allows you to define an alias for the package name to make your code easier to read and write. For example:

  ```
  import matplotlib.pyplot as plt
  plt.figure(1)
  ...
  ```

- `from <package> import <object1>,<object2>,...`. This allows only selected items from a package to be imported into the local workspace. All other contents of the package will be unavailable:

  ```
  from scipy.optimize import fsolve
  fsolve(...)
  ```

NOTES:

• You may also mix and match import methods, for example: '`from numpy import array as arr`' is legal python code.

• **It is not recommended to use** `from <package> import *`. While convenient, this can result in function overwriting, namespace pollution and is considered bad practice in general.

• Any modules you write yourself will be importable using these techniques provided that they are in the same directory.

2.2 Printing and Formatting

The python `print` function is used to display output to the user. The standard method of formatting a string prior to python3.6 was the `str.format` method shown below:

```
from math import pi
x = pi
print("x = {0:.2f}".format(x))
```

Once run this will produce the output: **x = 3.14**. The curly brackets denote a format specifier, the leading index refers to the corresponding index in the `format()` method, .2 means we are interested in two decimal places, and *f* means 'print as float'. We could also have used *e* 'scientific notation' or *g* 'automatic'. We may also modify the specifiers with alignments: < 'left aligned', > 'right aligned' and ∧ 'centre aligned' An example of these in practice is shown below:

```
print(" {0:<15.2f}, {2:>15.3g}, {1:^15.2e}".format(1/6, 143793.34850, 3.1415926))
```

In python3.6 and above, a new method of formatting (f-strings) was introduced. This allows variables to be embedded directly in strings, for example:

```
from math import pi
x = pi
print(f"x = {x:.2f}")
```

Produces **x = 3.14**.

2.3 Mathematical Operations

Python supports a number of basic mathematical operations using inbuilt operators such as ‘*’, ‘/’ etc.

Let `x = 2.5 ; y = 3.5`

Table 2: Basic Mathematical Operations in python

Operation	**Usage**	**Result**	**Operation**	**Usage**	**Result**
Addition	`z = x + y`	`z = 6.0`	Subtraction	`z = x - y`	`z = -1.0`
Multiplication	`z = x * y`	`z = 8.75`	Division	`z = y/x`	`z = 1.4`
Exponentiation	`z = x ** y`	`z = 24.705294220065465`			
Floor Division	`z = x // y`	`z = 0.0`	Modulus	z = x % y	`z = 2.5`
Is equal	`z = x == y`	`z = False`	Is not equal	`z = x != y`	`z = True`
Is less than	`z = x < y`	`z = True`	Is greater than	`z = x > y`	`z = False`

There is also a ‘is greater than or equal to’ >= and ‘is less than or equal to’ <= operator.

In addition to all of these, there exists in place operations, these are defined for all basic operations (except for equality and inequality checks). As an example:

```
>> x = 2.5 ; y = 3.5
>> x+= y
>> print(x)
6.0
```

This is a compact way of writing `x = x + y`.

Python also supports complex numbers, and most of the aforementioned operations on complex numbers (modulus, floor and inequality checks are not defined for complex numbers):

```
>> u = 3 + 5.j ; v = 6 - 2.j
>> u * v
(28+24j)
>> print(f"Re(u) = {u.real}, Im(u) = {u.imag}, u conjugate = {u.conjugate()}")
Re(u) = 3.0, Im(u) = 5.0, u conjugate = (3-5j)
>> u *= v
>> u
(28+24j)
```

NOTES:

• In python3.6 and above, underscores may be inserted into numbers to improve readability, e.g. `x = 345_298_531.45`.

• To convert explicitly between types, use `x = float(3) ; y = int(2.1)` etc. taking a float to an integer will perform a floor operation on the original number.

• Raising a negative number to a non-integer power using `**` will result in a complex number. The `math.pow` function will return an error instead.

• The math library functions are typically optimised for efficiency and should be used where possible, e.g `math.sqrt(2)` instead of `2**0.5`.

• The double precision floating point type `float` has a maximum relative accuracy[2] of approximately 10^{-16}. This is sufficient for almost all engineering applications. If greater precision is required then the `decimal` module may be used, and provides arbitrary precision.

[2]Meaning that the result of two floating point operations will be expected to be accurate to 15 significant figures. This **does not** mean that the smallest number which can be represented by double precision floating points is 10^{-16}, numbers as small as 2.23×10^{-308} can be represented

2.4 Lists

A list in python is an array of information, lists can include mixed data types. Lists can be invoked using square brackets or using the `list()` function:

```
>> a = [15, 16, 17, 18, 19] # defining a list
```

Lists can be indexed and sliced using square brackets, for example:

```
>> a[0] # returns the first element of a
15
>> a[2] # returns the third value of a
17
>> a[-1] # returns the last value of a
19
>> a[2:4] # returns the 3rd to 4th values of a
[17,18]
>> a[::2] # returns every second element of a
[15, 17, 19]
>> a[::-1] # returns the list a reversed
[19, 18, 17, 16, 15]
```

Lists can contain mixed data types, including other lists, most python data types, and even functions.

```
>> b = [1, '2', 3.5, 4 + 5.j, [6, 7], sum] # A valid list definition
```

Values can be added to the end of a list using the `append` and `extend` methods:

```
>> l1 = [1, 2]
>> l2 = [3, 4]
>> l1.extend(l2)
>> print(l1)
[1, 2, 3, 4]
>> l1.append(5)
>> print(l1)
[1, 2, 3, 4, 5]
```

NOTES:

• All fundamental python3 data types including lists start indexing at 0.

• Lists should NOT be used for linear algebra or large amounts of calculations. For these applications, the `numpy.array` is better suited and much more efficient.

• Lists should primarily be used to store similar data types in an ordered way and to append data dynamically.

• You can get the length of a list called 'list_name' using `len(list_name)`

• To copy a list safely, use: `new_list = list(old_list)`

• The data type `tuple`, is mostly identical to a list except that the entries are *immutable* (cannot be modified).

2.5 Dictionaries

Dictionaries are key value mappings. They are particularly useful for storing and retrieving information which has a logical grouping. Dictionaries are invoked using curly brackets:

```
>> particle = {'x': 1.0, 'y': 2.0, 'z': -1.0, 'vx': 0.0, 'vy': -2.0, 'vz': 3.0}
>> print(particle['y'])
2.0
```

In this case, the 'key' is the string `'x'` and the value is 2.0. Key/value pairs can be added or removed or overridden from existing dictionaries:

```
>> particle['mass'] = 5.0 # adds a key 'mass' with value 5.0
>>print(particle['mass'])
5.0
>>del particle['mass'] # delete 'mass' and its value from the dictionary
>>particle['vx'] = 1.0 # changes the value of the keyword 'vx' to 1.0
>> print(list(particle.items())) # prints out a list of the key/value pairs
```

2.6 Sets

Sets are unordered collections of unique data, they are invoked using curly braces or the `set()` function. Sets have no order and cannot be idexed.

```
>> s = {1, 2, 3, 4, 4, 5}
>> print(s)
{1, 2, 3, 4, 5} # duplicate 4 has been removed
>> s2 = {4, 5, 6, 7, 8}
>> s & s2 # intersection operator
{4, 5}
>> s | s2 # union operator
{1, 2, 3, 4, 5, 6, 7}
>> s - s2 # complement operator
{1, 2, 3}
```

Sets provide an easy way to check if every element in a list is unique

```
>> a = [1,2,3,3,4] # make a list
>> len(set(a)) == len(a) # test if every element in a is unique
False
```

They are also convenient to check if an input is one of multiple valid values without using lengthy if statements:

```
x = 'a'
if x in {'a', 'b', 'c'}:
    do_something()
else:
    do_something_else()
```

2.7 For Loops

For loops iterate over every value in some data structure, and perform a given operation using that value. They are essential for performing repetitive actions on many different inputs:

```
# A loop which iterates over the values 5-9
for i in range(5, 10):
    print(i)

# A loop which iterates over a list of complex numbers
a = [1 + 1.j, 3 - 4.j, 5 + 6.j]
for x in a:
    print(x)
```

Sometimes both the value of some iterator and its index may be required, the `enumerate` generator provides a convenient way to do this:

```
a = [10,12,16,18,20]
for i,x in enumerate(a):
    print(i,x)
```

Or if it is required to iterate over multiple sets of data, use the `zip(data1, data2,...)` generator:

```
a = [10,12,16,18,20]
b = [-10,-5,0,5,10]
for x,y in zip(a,b):
    print(x,y)
```

Loops can also be nested within other loops:

```
for i in range(0,5):
    for j in range(0,5):
        print(i,j)
```

Sometimes it is convenient to write a simple loop or summation in a single line. This can be done using *list comprehension*.

```
>> odds = [2*n+1 for n in range(10)] # get the first 10 odd numbers using list comprehension
>> print(odds)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>> basel = sum(1/(i*i) for i in range(1,100)) # sum up the first 100 terms of the Basel sum
>> print(basel)
1.6348839001848923
```

2.8 While Loops

A while loop will perform some iterative action while some condition is `True`.

```
import random # the random number module
counter = 0
amount  = 1000
while amount > 0:
    amount -= random.random() # subtract some random value between 0 and 1 from amount
    counter += 1
print(f"It took {counter} iterations for the amount to become negative")
```

A while loop will perform an iterative action while `condition == True`. *condition*, may be any valid python boolean comparison.

```
while condition:
    do_something()
```

NOTES

• `while` loops are best used for iterative actions where the number of iterations required will not be known *a priori*. If the iteration will require a known in advance (and fixed) number of iterations then a `for` loop may be more appropriate.

2.9 Functions

Functions are objects which can accept zero or more inputs, and which perform an action or series of actions when used. They are best used to cluster repetitive actions into a single line of code and eliminate repetition. Functions are defined using the `def` statement, and can return zero or more inputs using the `return` statement. As an example, consider the function which computes the area of a circle `circle_area`

```
import math
def circle_area(r):
    "A function which calculates the area of a circle of radius r"
    return math.pi*r*r
```

The above function is supplied the input ‘r’ and returns the circle area. The string inside the function is a special kind of comment called a *doc-string*. This doc string will be displayed if we use the inbuilt `help` function.

```
>> help(circle_area)
```

```
Help on function circle_area in module __main__:

circle_area(r)
    a function which computes the area of a circle of radius r
```

```
>> A = circle_area(10) # an example of using the circle_area function with a radius of 10
>> print(A)
314.1592653589793
```

Functions can have multiple inputs:

```
def cylinder_volume(r, h):
    return math.pi*r*r*h

>> cylinder_volume(2, 6)
75.39822368615503
```

They can also have no inputs and/or no return value:

```
def print_pi():
    print(f"pi is approximately: {math.pi}")

>> print_pi()
"pi is approximately: 3.141592653589793"
```

Functions may also have a default value:

```
def line(m, x, c = 0):
    return m*x + c

>> line(4,1) # no input given for c, uses default value of 0
4
>> line(4,1,c=-1) # c assigned as -1, default value overridden
3
```

It is also possible to pass in a variable number of arguments, and keyword arguments using the * or ** prefix:

```
def prod(*vals):
    "returns the product of all given values"
    result = 1.0
    for val in vals:
        result *= val
    return result

>> prod(3.1, 4, 5.6)
69.44
```

A similar thing can be done with keyword arguments:

```
def format_results(**results):
    "prints out a list of values in an aligned format"
    format_string = "{0:>20} : {1:<20}"
    for result in results:
        print(format_string.format(result, results[result]))

>> format_results(x = 1, y = 2, z = 3, w = -4, k = 7)
```

```
                    x  :  1
                    y  :  2
                    z  :  3
                    w  :  −4
                    k  :  7
```

Functions can also accept other functions as input:

```
def central_difference(f, x, h = 1.0E-4):
    "Take a central difference approximation of the derivative of f(x) at x"
    return (f(x + h) - f(x - h))/(2.0*h)

def func(x):
    "test function"
    return 3*x*x

>> central_difference(func, 1.0)
6.00000000000156
```

Finally, functions can return other functions as output, this is called a *closure*:

```
def generate_polynomial(*coeffs):
    "Returns a callable polynomial function with coefficients (coeffs)"
    def poly(x):
        return sum(coeff*x**i for i,coeff in enumerate(coeffs))
    return poly

>> quadratic = generate_polynomial(1, 2, -1) # define the quadratic 1 + 2x - x^2
>> quadratic(3.1) # evaluate that quadratic at x = 3.1
-2.41
```

NOTES:

• Functions can use variables assigned in the module level workspace, but the reverse is not true, variables defined within the function do not exist outside of it, all information other than the return value is automatically destroyed. As an example:

```
a = 1
def f(x):
    b = 2
    return a*x + b # a is accessible from outside the function

>> print(b) # Fails with error because b only exists within the function
```

• Despite the above point, it is generally considered bad practice for a function to rely on external variables, although constants defined at the module level are sometimes acceptable. If constants need to be passed into a function it is preferred to use closures or classes.

• A function which contains a combination of positional, default, variable arguments, and variable keyword arguments takes the form `def func(args, default_args = default, *args, **kwargs)`.

2.10 Conditional (if) Statements

`if, elif, else` statements are used to control the flow of your code based on a series of possible conditions. The `if condition` statements first tests whether `condition == True`. If that check fails, then all `elif condition` statements will be tested in order of listing until one passes. The remaining statements are ignored. If all `if` and `elif` statements fail, then the code will proceed onto the `else` statement which will always execute if it is reached. It is not required to include an `else` statement. If the `if` and all `elif` statements fail and there is no `else` statement then the code will exit without performing an action. Consider the following example of a piecewise function.

```
def piecewise_function(x):
    if x < 0:
        return -x
    elif 0 <= x <= 4:
        return x
    else:
        return 0.25*x*x - (4 - x)
```

NOTES:

• It is legal to have a single `if` statement on its own without any `elif` or `else` statement.

• Having multiple `if` statements after each other will test every statement regardless of whether each passes. This can lead to incorrect results if you are testing cases and intended to use an `elif` instead.

• `if, elif, else` statements can be nested within other `if, elif, else` statements for more complex control flow problems.

3 numpy and Linear Algebra

numpy is a package designed for efficiently operating with multidimensional arrays. **Any linear algebra or large scale data processing performed in python should be done using the numpy package**.

3.1 Basic Array Operations

```
import numpy as np
```

Defining numpy Arrays: Arrays can be manually defined and populated using the array object. Otherwise, preset arrays of certain sizes and configurations may be defined using various numpy modules such as linspace, logspace, empty, zeros, identity etc.

```
# define a vector of double precision floats
b = np.array([1, 2, 3], float)
# define a two dimensional array
A = np.array([[3, -4, 2],
              [-1, 4, 3],
              [0, -7, 6]], float)
# generate an array of 100 linearly spaced values between -pi and +pi
a = np.linspace(-np.pi, np.pi, 100)
# generate an array of log-spaced values
m = np.logspace(-8,-1,8)
# generate an empty array (this is useful to reserve the data container before
# populating it with data)
B = np.empty((4,4)) # define an empty 4 by 4 2d array
```

Indexing, slicing, and copying arrays: An array may be *indexed* to retrieve or view the value of its elements at a certain position.

```
>> print(A[1,2]) # prints the 2nd row, 3rd column element of A
3.0
>> b[1] = 4 # reassign the 2nd element of b as 4.0
>> print(b)
array([ 1.,  4.,  3.])
>> print(B.shape) # get the number of (rows, columns) of B as a tuple
(4,4)
```

Slicing refers to the act of retrieving a subsection of the array to perform some operation on it:

```
>> A[:,1] = [-3, 0, 1] # replaces the 2nd column of A with [-3, 0, 1]
>> print(A)
array([[ 3., -3.,  2.],
       [-1.,  0.,  3.],
       [ 0.,  1.,  6.]])
>> B[...] = np.pi # replace every element of B with pi
>> print(B)
array([[ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265]])
>> B[2:4,2:4] = 0.0 # replace the 2 by 2 lower right corner of the array with 0.0
>> print(B)
array([[ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  0.        ,  0.        ],
       [ 3.14159265,  3.14159265,  0.        ,  0.        ]])
```

To *copy* an array, use `numpy.copy`

```
b2 = np.copy(b) # create a copy of b called b2
row1 = np.copy(B[0]) # create a 1d array by copying the first row of B
```

IMPORTANT!: It may be tempting to 'copy' an array using a simple assignment such as:

```
b3 = b
```

However **this does not create a true copy of b, instead it creates a reference to b called b3**. This can be observed if we modify b in some way:

```
>> b[2] = 0 # set the 3rd element of b to zero
>> print(b)
array([ 1.,  4.,  0.])
>> print(b2)
array([ 1.,  4.,  3.]) # the copy is unaffected
>> print(b3)
array([ 1.,  4.,  0.]) # the reference is modified along with b
```

Both operations can be useful, but will lead to errors if used incorrectly.

3.2 Linear Algebra

numpy supports many linear algebra routines, some examples are shown in the following code block.

```
# define some testing arrays
b = np.array([1, 2, 3], float)
A = np.array([[3, -4, 2],
              [-1, 4, 3],
              [0, -7, 6]], float)
# dot product of two vectors
u = b @ b
# matrix/matrix multiplication
C = A @ A
# matrix/vector multiplication
v = A @ b
# element wise basic math on arrays (supports same operations as python math)
w = v + u ; w = v - u; w = v * u ; w = v/u  #etc
# scalar multiplication by array
R = 2.5*A
# matrix transpose
D = A.T
# array operations in place
A += C ; A -=C # etc, same as python math
# solve linear system A*x = b
x = np.linalg.solve(A, b)
```

NOTES:

• In place array operations are more efficient, as a new array does not have to be created every time an operation is performed.

4 Solving Non-Linear Equations

The `scipy` package contains optimisation and root finding algorithms which are useful to solve systems of non-linear equations. The `scipy.optimize.fsolve` function is sufficient for most applications. To use these functions, you must first define the system of equations in the form $f(x) = 0$.

Example: solve the system of equations:

$$\begin{aligned} x^2 - y^2 &= 0 \\ xy &= 2 \end{aligned}$$

```
from scipy.optimize import fsolve
```

Define the system of equations to solve:

```
def zero_function(X):
        x,y = X
    return [x**2 - y**2,
            x*y - 2]
```

The first input to `fsolve` must be the system of equations to solve, and the second an initial guess for the value of the root:

```
root = fsolve(zero_function,[1,1])

>> print(root)
array([ 1.41421356,  1.41421356])
```

NOTES:

- `fsolve` will only find a single solution, in the above example, there is a second solution at $(x,\ y) = (-\sqrt{2},\ -\sqrt{2})$. A careful choice of initial guess will usually find the solution of interest.

5 matplotlib and Plotting

`matplotlib`[3] is the standard open source plotting package for python. It can be used to create a wide variety of plot types, including: 2d and 3d plots, heat maps, contour maps, basic 2d and 3d graphics, and even simple animations.

5.1 Plotting in 2 dimensions

A basic 2d plot in matplotlib takes the following form:

```python
import matplotlib.pyplot as plt
plt.figure()
plt.plot([1,2,3], [2,-4,6], '-r')
plt.show()
```

In the above example, `plt.figure()` defines the creation of a new plot object, `plt.plot(x, y, style)` plots x vs y using some 'style', and `plt.show()` displays the plot and clears data for the next plot object.

A more advanced example showing off some useful features of matplotlib including sub-plots and mathematical formatting is attached below:

```python
import matplotlib.pyplot as plt
import numpy as np

# make up some data to plot
x_data = np.linspace(-np.pi, np.pi, 100)
y_data = np.sin(x_data)
y_data2 = np.sin(x_data**2)

# define a figure, and two sub-plots of that figure and assign them to a shared
# x axis
fig, (ax1, ax2) = plt.subplots(nrows = 2, sharex = True)

# configure the first sub-plot
ax1.plot(x_data, y_data, '-k', label = r'$ \sin \ \lambda$')
ax1.set_ylabel(r'$y_1 (\lambda)$')
ax1.grid(True)
ax1.legend()

# configure the second sub-plot [::3] is a numpy slicing which takes
# every 3rd point
ax2.plot(x_data, y_data2, '-k')
ax2.plot(x_data[::3], y_data2[::3], '.k', label = r'$ \sin \ \lambda^2$')
ax2.grid(True)
ax2.legend()
ax2.set_ylabel(r'$y_2 (\lambda)$')
ax2.set_xlabel(r'$\lambda$')

# display the result
plt.show()
```

Once executed, Figure 1a is produced:

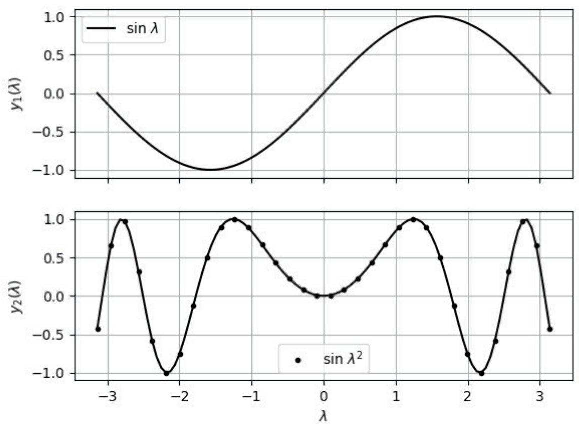
5.2 Plotting in 3 dimensions

An example of a surface plot in 3 dimensions is given below. The general idea is to use numpy to generate a `np.meshgrid` object and map the data of interest to this object. Other formats are possible, refer to the matplotlib documentation (https://matplotlib.org/).

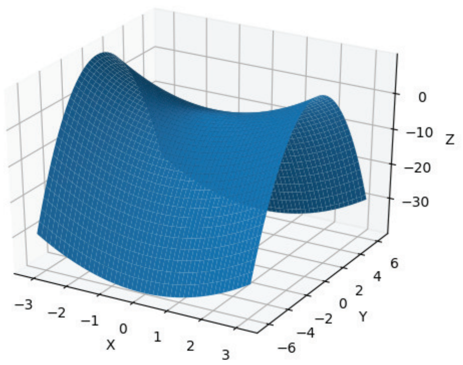
[3]https://matplotlib.org/

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the meshgrid
X, Y = np.meshgrid(
        np.linspace(-np.pi, np.pi, 50),
        np.linspace(-2*np.pi, 2*np.pi, 100)
)

# make some Z data over that meshgrid
Z = X*X - Y*Y

# create the plot
fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.plot_surface(X, Y, Z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```

Once executed, Figure 1b is produced.



(a)



(b)

Figure 1: matplotlib can produce lots of nice figures!

NOTES:

• There is a subset of `matplotlib` modelled on matlab called `pylab`, however, it has largely been depreciated and it is recommended that users stick to the core `matplotlib` package for recent versions of python.

• matplotlib is not a 3d graphics engine, for more sophisticated 3d graphics in python, check out the `mayavi` package.

6 Saving and Loading Data from and to file

6.1 Loading a .txt file of numeric data

If the text file contains only numerical data, it can be loaded very easily using numpy:

```
import numpy as np
data = np.loadtxt('filename.txt')
```

`data` will be an array with the correct number of columns and rows.

6.2 Saving and Loading numpy arrays

numpy provides powerful tools for saving and loading data, you may use the `numpy.save` functionality:

```
import numpy as np
# make an array of random numbers with 10 rows and 4 columns
A = np.random.random((10,4))
np.save('data',A)
```

This will create a *numpy binary data file* in the current directory called 'data.npy'.

To load this data back into python use `numpy.load`

```
B = np.load('data.npy')
```

Notes:

- To read and write general file types including text files and other python files, use the python `open` function [4] .

- There is also a `csv` module designed to read and write .csv files.

[4] A decent tutorial can be found here: http://www.pythonforbeginners.com/files/reading-and-writing-files-in-python

7 Recommended Module Structure and Style Convention

7.1 Recommended Module Layout

```
"""
Module should begin with a document string which outlines the purpose of the
module
"""

# import statements should be at the top
import module1
import moduel2
...

# define module level constants next
CONSTANT_1 = ...
CONSTANT_2 = ...
...

# next comes all function/class/iterator definitions
def function_1(inputs, ...):
        "function 1 doc string"
        ...

def function_2(inputs, ...):
        "function 2 doc string"
        ...

# can then define a main function which contains the procedural operations of
# the code, this is optional in python, and some programmers choose to leave it
# out. Does not require a return statement unless you want to assign the output
# of your program to something
def main():
        ...

# finally, the following statement will run the main function when the module
# is executed. If you did not write a main() function, then can put the
# procedural operations of your code here
if __name__ == '__main__':
        main()
```

What is the advantage of doing this?
It makes the logic of longer programs more clear and easier to debug. The main procedure of the program takes place in main(). If there is a bug in a certain function then the programmer only needs to look at that particular function definition rather than dealing with complex control flow. The workspace is not cluttered with (potentially conflicting) variable declarations. Constant definitions are all in the same place.

7.2 Recommended Programming Conventions

The formal python style guide as of writing is *Python Enhancement Proposal 8* (pep8) [5]. This is a set of recommendations for writing consistent and readable code. It is not enforced, but is considered good practice to adhere to this guide when practical.

Why should we use a programming style convention? Code is read more often than it is written, and the primary reader of your code is yourself. Teaching yourself good programming conventions can significantly improve your work speed, improve the readability of your code, make it easier to debug, and make it easier for others to read and contribute to. A few of the most useful recommendations from pep8 for engineering and scientific applications include:

[5]https://www.python.org/dev/peps/pep-0008/

Use descriptive variable names over short variable names: Unlike in handwritten mathematics, code can easily be copied and pasted, hence long but descriptive variable names are preferred over short but obtuse ones. Variable and function names should be lower-case, and separated by underscores. Constant names should be all caps and separated by underscores.

```
# constants should be in upper-case and separated by underscores
MASS_EARTH      = 5.972E24      # mass of earth [kg]
MASS_SATELLITE = 1000          # mass of satellite [kg]
G               = 6.67408E-11  # gravitational constant [m3/kg.s2]
MU_EARTH        = MASS_EARTH*G # earth standard gravitational parameter [m3/s2]

# variables should be in lower-case and separated by underscores
satellite_pos = np.array([20000E3, - 15000E3, 1000E3])
satellite_radius = np.linalg.norm(satellite_position)
satellite_force = -MU_EARTH*MASS_SATELLITE/(satellite_radius**3)*satellite_pos
```

Split long calculations onto multiple lines: Parenthesis can be used to extend lines, when you are unable to fit a long calculation onto a single line.

```
long_calculation = (coeff_1*var_1   + coeff_2*var_2**2
                 + coeff_3*var_3**3 - coeff_4*math.exp(var_3)
                 + ...
)
```