Python-Summary-181204

December 4, 2018

1 Introduction to Computational Physics - Python summary

1.1 Table of contents

2 Importing libraries

2.1 Numpy and matplotlib

```
In [96]: # <!-- Student -->
         # Numpy is a Python library of numerical routines
         import numpy as np
         #
         # Matplotlib.pyplot is a library of plotting routines
         import matplotlib.pyplot as plt
         #
         # This "line magic" ensures matplotlib plots are displayed in the Notebook
         %matplotlib inline
```

2.2 Sympy and least_squares

```
In [97]: # <!-- Student -->
         # sympy is the python sympbolic manipulation library
         import sympy as sp
         #
```

```
        # least_squares is a fitting routine
        from scipy.optimize import least_squares
```

3 Calculations with Python

```
In [98]: # <!-- Student -->
         A = 1 + 4 # addition
         B = 3 - 8 # subtraction
         C = 4*3   # multiplication
         D = 3/4   # division
         E = 2**10 # raise to power
         F = 17//5 # integer division
         G = 17%5  # remainder after division
         print("1+ 4 =",A)
         print("3 - 8 =",B)
         print("4*3 =",C)
         print("3/4 = ",D)
         print("2**10 =",E)
         print("17/5 =",F)
         print("17%5 =",G)
```

```
1+ 4 = 5
3 - 8 = -5
4*3 = 12
3/4 =  0.75
2**10 = 1024
17/5 = 3
17%5 = 2
```

4 Assigning variable names to quantities

Names can be bound to quantities using the "=" sign.

```
In [99]: # <!-- Student -->
         r = 0.2
         pi = 3.1415927
         A = pi*r**2
         print("Area of circle is",A)
```

```
Area of circle is 0.125663708
```

5 Functions in Python

A large range of functions is provided in numpy, see numpy mathematical functions. Use as in examples below.

```
In [100]: # <!-- Student -->
          angle = 1.42 # radians
          sinAngle = np.sin(angle)
          x = 176.4
          logx = np.log(x) # logarithm to the base e
```

Note, the functions `np.sin`, `np.cos` etc. expect their arguments to be in radians, not degrees.

6 Defining functions

Functions can be defined as follows:

```
In [101]: # <!-- Student -->
          def rectPrismParams(w, l, h):
              '''
              Return volume, surface area and length of edges of rectangular prism
              given its width, length and height
              '''
              V = w*l*h
              A = 2*(w*l + w*h + l*h)
              s = 4*(w + l + h)
              return V, A, s
          #
          w = 0.3
          l = 0.17
          h = 0.25
          #
          V, A, s = rectPrismParams(w, l, h)
          #
          print("Rectangular prism width",w,"length",l,"height",h)
          print("Volume",V)
          print("Surface area",A)
          print("Length of edges",s)
```

```
Rectangular prism width 0.3 length 0.17 height 0.25
Volume 0.012750000000000001
Surface area 0.337
Length of edges 2.88
```

7 Python data types

Numerical data types include:

- Integers (e.g 57, -163).
- Real numbers (e.g. 57.0, 5.7E1 or -5.7e1), which are referred to as `floats` in Python.
- Complex numbers (e.g. 57.0 - 75.0j).

Further useful data types are:

7.1 Boolean variables

These have the value `True` or `False`. Their values can be directly assigned or can result from e.g. comparing two numbers.

```
In [102]: # <!-- Student -->
          boolVar1 = False
          print("boolVar1",boolVar1)
          boolVar2 = 3 > 2
          print("boolVar2",boolVar2)
          boolVar3 = 2 > 3
          print("boolVar3",boolVar3)
```

```
boolVar1 False
boolVar2 True
boolVar3 False
```

Numbers can be compared in a variety of ways:

```
In [103]: print("2 > 3",2 > 3) # greater than
          print("2 == 2",2 == 2) # equal to
          print("2 >= 2",2 >= 2) # greater than or equal to
          print("3 <= 2",3 <= 2) # less than or equal to
          print("2 != 2",2 != 2) # not equal to
```

```
2 > 3 False
2 == 2 True
2 >= 2 True
3 <= 2 False
2 != 2 False
```

Logical operators can be used in conjunction with boolean variables.

```
In [104]: # <!-- Student -->
          print("True and True =",True and True)
          print("not True =",not True)
          print("False or True =",False or True)
```

```
True and True = True
not True = False
False or True = True
```

7.2 Strings

Strings (e.g. “&”, " “,”Much longer string with several characters¡‘) consist of one or more of the letters, numbers and symbols on your keyboard, delimited by”" or ”.

```
In [105]: # <!-- Student -->
          testString = "Mary had a little lamb"
```

Python offers a wide range of functions for working with strings for example:

```
In [106]: # <!-- Student -->
          findMary = "Mary" in testString
          findmary = "mary" in testString
          findSpace = " " in testString
          print("findMary =",findMary)
          print("findmary =",findmary)
          print("findSpace =",findSpace)
```

```
findMary = True
findmary = False
findSpace = True
```

```
In [107]: # <!-- Student -->
          indexLamb = testString.index("lamb")
          indexSpace = testString.index(" ")
          print("indexLamb",indexLamb) # Returns index at which string searched for starts
          print("indexSpace",indexSpace) # Returns first instance
```

```
indexLamb 18
indexSpace 4
```

```
In [108]: # <!-- Student -->
          numAs = testString.count("a")
          print("Number of 'a's in testString is",numAs)
```

```
Number of 'a's in testString is 4
```

```
In [109]: # <!-- Student -->
          print("testString[6] =",testString[6])
```

```
testString[6] = a
```

```
In [110]: # <!-- Student -->
          testString = testString.replace("little", "huge")
          print("testString",testString)
```

```
testString Mary had a huge lamb
```

7.3 Lists

Lists can be defined as follows:

```
In [111]: # <!-- Student -->
          thisList = [1, 2.278, -8, "cheese", 'blue ', np.pi]
          emptyList = []
          print("thisList =",thisList)
          print("emptyList =",emptyList)
```

```
thisList = [1, 2.278, -8, 'cheese', 'blue ', 3.141592653589793]
emptyList = []
```

The entries in a list can be accessed using their indices.

```
In [112]: # <!-- Student -->
          print("thisList[0] =",thisList[0])
```

```
thisList[0] = 1
```

Lists are mutable, i.e. the entries in a list can be changed.

```
In [113]: # <!-- Student -->
          thisList[0] = 3.0
          print("thisList",thisList)
```

```
thisList [3.0, 2.278, -8, 'cheese', 'blue ', 3.141592653589793]
```

Entries can be appended to a list.

```
In [114]: # <!-- Student -->
          appendReturn = emptyList.append("something")
          print("emptyList",emptyList)
          print("Note that append returns",appendReturn)
```

```
emptyList ['something']
Note that append returns None
```

Where the returned value is `None`, the method can be used without a return value as follows.

```
In [115]: # <!-- Student -->
          emptyList.append("something else")
          print("emptyList",emptyList)
```

```
emptyList ['something', 'something else']
```

Lists can be added (using +) or extended. Note the difference to append. Addition or extension result in the elements of the list being added to the original list, whereas append adds the appended list as a single element.

```
In [116]: # <!-- Student -->
          thatList = ["new", 1]
          print("thisList + thatList =\n",thisList + thatList)
          thisList.extend(thatList)
          print("thisList =\n",thisList)
```

```
thisList + thatList =
 [3.0, 2.278, -8, 'cheese', 'blue ', 3.141592653589793, 'new', 1]
thisList =
 [3.0, 2.278, -8, 'cheese', 'blue ', 3.141592653589793, 'new', 1]
```

Elements can also be “popped” out of a list (in which case they are no longer in the list).

```
In [117]: # <!-- Student -->
          popOut = thisList.pop(3)
          print("popOut",popOut)
          print("thisList",thisList)
```

```
popOut cheese
thisList [3.0, 2.278, -8, 'blue ', 3.141592653589793, 'new', 1]
```

Elements can be inserted in a list.

```
In [118]: # <!-- Student -->
          insertReturn = thisList.insert(1, "pinkish")
          print("thisList = ",thisList)
```

```
thisList =  [3.0, 'pinkish', 2.278, -8, 'blue ', 3.141592653589793, 'new', 1]
```

7.4 Tuples

Tuples are similar to lists but are immutable.

```
In [119]: # <!-- Student -->
          thisTuple = (3.14, -9, "orange")
          print("thisTuple[2] =",thisTuple[2])
```

```
thisTuple[2] = orange
```

7.5 Identifying types and type conversion

This can be done for some of the more common cases as follows:

```
In [120]: # <!-- Student -->
          #
          print("Type of thisTuple is",type(thisTuple))
          print("Type of 3",type(3))
          print("Type of False",type(False))
```

```
Type of thisTuple is <class 'tuple'>
Type of 3 <class 'int'>
Type of False <class 'bool'>
```

```
In [121]: # <!-- Student -->
          #
          intFalse = int(False)
          intTrue = int(True)
          floatFalse = float(False)
          floatTrue = float(True)
          print("intFalse = ",intFalse)
          print("intTrue = ",intTrue)
          print("floatFalse = ",floatFalse)
          print("floatTrue = ",floatTrue)
```

```
intFalse =  0
intTrue =  1
floatFalse =  0.0
floatTrue =  1.0
```

8 Pretty printing

The legibility of printed output can be improved by using the format statement.

```
In [122]: # <!-- Student -->
          big = 999.999
          small = 666.666E-19
          integer = 33
          string = 'letters'
          print("Big is {:5.2f}, small is {:3.3e}, integer is {:d} and string is {:s}!"
                .format(big, small, integer, string))
```

```
Big is 1000.00, small is 6.667e-17, integer is 33 and string is letters!
```

```
In [123]: # <!-- Student -->
          print("Backspacing is \bpossible, as is moving to a \nnew line.")
```

```
Backspacing ispossible, as is moving to a
new line.
```

Using “tab” characters (\t) can help with formatting (though you might have to do some tweaking to get things to line up as you want). A tab is eight characters long.

```
In [124]: # <!-- Student -->
          print("Months in the year:")
          print("1 \t 2 \t 3 \t 4 \t 5 \t 6 \t 7 \t 8 \t 9 ")
          print("Jan \t Feb \t Mar \t Apr \t May \t Jun \t Jul \t Aug \t Sept")
```

```
Months in the year:
1          2          3          4          5          6          7          8          9
Jan          Feb          Mar          Apr          May          Jun          Jul          Aug
```

Note, there are lots more tools for writing things prettily, see here, for example.

9 Numpy arrays

9.1 One dimensional arrays

One dimensional arrays (vectors) can be created and assigned values in various ways, as is shown below.

```
In [125]: # <!-- Student -->
          arrayFromList = np.array([0, 10, 20, 20, 40])
          #
          length = 3
          arrayOfZeros = np.zeros(length)
          arrayOfOnes = np.ones(length)
          arrayOfSixes = np.full(length, 6.0)
          #
          arrayPi = np.zeros(length)
          arrayPi[0] = 3
          arrayPi[1] = 1
          arrayPi[2] = 4
          #
          print("arrayFromList",arrayFromList)
          print("arrayOfZeros",arrayOfZeros)
          print("arrayOfOnes",arrayOfOnes)
          print("arrayPi",arrayPi)
          print("arrayOfSixes",arrayOfSixes)
```

```
arrayFromList [ 0 10 20 20 40]
arrayOfZeros [0. 0. 0.]
arrayOfOnes [1. 1. 1.]
arrayPi [3. 1. 4.]
```

```
arrayOfSixes [6. 6. 6.]
```

```
In [126]: # <!-- Student -->
          length = 3
          start = 10
          stop = 30
          arrayOfNumbers = np.linspace(start, stop, length)
          print("arrayOfNumbers",arrayOfNumbers)
```

```
arrayOfNumbers [10. 20. 30.]
```

```
In [127]: # <!-- Student -->
          beg = 0.0
          end = 20.0
          nSteps = 10
          stepArray = np.linspace(beg, end, nSteps + 1)
          print("stepArray =",stepArray)
```

```
stepArray = [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18. 20.]
```

The last (and next-to-last, and next-to-next-to-last) elements of the array can be accessed by using the indices -1 (and -2, and -3).

```
In [128]: # <!-- Student -->
          #
          print("stepArray[-1] =",stepArray[-1])
          print("stepArray[-2] =",stepArray[-2])
          print("stepArray[-3] =",stepArray[-3])
```

```
stepArray[-1] = 20.0
stepArray[-2] = 18.0
stepArray[-3] = 16.0
```

9.2 Two and more dimensional arrays

Numpy arrays with two or more dimensions can also be created. The dimensions are defined by a tuple, e.g. `(3, 4)` for a matrix with three rows and four columns.

```
In [129]: # <!-- Student -->
          matrixOnes = np.ones((3, 4))
          matrixSixes = np.full((3, 4), 6.0)
          print("matrixOnes =\n",matrixOnes)   # \n in starts a new line!
          print(" ")
          print("matrixSixes =\n",matrixSixes)
```

```
matrixOnes =
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

matrixSixes =
 [[6. 6. 6. 6.]
 [6. 6. 6. 6.]
 [6. 6. 6. 6.]]
```

9.3 Shape or length of an array

The shape or length of an array can be determined as follows:

```
In [130]: # <!-- Student -->
          print("matrixSixes.shape =",matrixSixes.shape)
          print("len(matrixSixes) =",len(matrixSixes))
```

```
matrixSixes.shape = (3, 4)
len(matrixSixes) = 3
```

Note that `len()` returns the length of the zero axis (the number of rows here) and is an integer. The result of `matrix.shape` is a tuple.

9.4 Slicing numpy arrays

We can look at ranges of elements in an array. Here is a 1D example:

```
In [131]: # <!-- Student -->
          countArr = np.linspace(0, 4, 5)
          print("countArr =",countArr)
          print("countArr[1:4] =",countArr[1:4])
```

```
countArr = [0. 1. 2. 3. 4.]
countArr[1:4] = [1. 2. 3.]
```

The syntax `countArr[iLow:iHigh]` returns the elements from `iLow` up to `iHigh`, including `iLow` but not `iHigh`. Using `countArr[iLow:iHigh:iStep]` returns the elements from `iLow` up to `iHigh`in steps of `nStep`. The last value returned is that with index within `nStep` of `iHigh`.

```
In [132]: # <!-- Student -->
          print("countArr[1:4:2] =",countArr[1:4:2])
```

```
countArr[1:4:2] = [1. 3.]
```

Similarly for 2D (and higher dimensional) arrays.

```
In [133]: # <!-- Student -->
          print("matrixSixes \n",matrixSixes)
          print(" ")
          print("matrixSixes[1:3, 0:2] \n",matrixSixes[1:3, 0:2])
```

```
matrixSixes
 [[6. 6. 6. 6.]
 [6. 6. 6. 6.]
 [6. 6. 6. 6.]]

matrixSixes[1:3, 0:2]
 [[6. 6.]
 [6. 6.]]
```

9.5 Maximum and minimum values of arrays

Maximum value in array along indicated axis. (Same syntax for minimum value.)

```
In [134]: # <!-- Student -->
          matrix = [[1,17,3], [4,0,6]]
          aMaxRow = np.max(matrix, 0)
          aMaxCol = np.max(matrix, 1)
          print("matrix =\n",matrix)
          print("aMaxRow =",aMaxRow)
          print("aMaxCol =",aMaxCol)
```

```
matrix =
 [[1, 17, 3], [4, 0, 6]]
aMaxRow = [ 4 17  6]
aMaxCol = [17  6]
```

Make array containing minimum values of elements of two arrays of same length. (Same syntax for maximum values.)

```
In [135]: # <!-- Student -->
          arrayUp = np.linspace(0, 5, 6)
          arrayDown = np.linspace(5, 0, 6)
          arrayMin = np.minimum(arrayUp, arrayDown)
          print("arrayUp",arrayUp)
          print("arrayDown",arrayDown)
          print("arrayMin",arrayMin)
```

```
arrayUp [0. 1. 2. 3. 4. 5.]
arrayDown [5. 4. 3. 2. 1. 0.]
arrayMin [0. 1. 2. 2. 1. 0.]
```

9.6 Sums and products of array elements

The contents of an array can be summed to give a single value, or an array containing the cumulative vaues of the sum. Slicing can be used here (as shown for 'np.prod' and 'np.cumprod' below) to select the elements that should be summed.

```
In [136]: # <!-- Student -->
          sumArrayUp = np.sum(arrayUp)
          cumSumArrayUp = np.cumsum(arrayUp)
          print("sumArrayUp",sumArrayUp)
          print("cumSumArrayUp",cumSumArrayUp)
```

```
sumArrayUp 15.0
cumSumArrayUp [ 0.   1.   3.   6. 10. 15.]
```

Product of contents of array can be taken to give a single value, or an array containing the cumulative values can be returned.

```
In [137]: # <!-- Student -->
          prodArrayUp = np.prod(arrayUp[1:6])
          cumProdArrayUp = np.cumprod(arrayUp[1:6])
          print("prodArrayUp",prodArrayUp)
          print("cumProdArrayUp",cumProdArrayUp)
```

```
prodArrayUp 120.0
cumProdArrayUp [  1.   2.   6.  24. 120.]
```

9.7 Boolean masking of arrays

Logic can be used to identify the required components of an array. These can be used to generate a new array conteining only the required components.

```
In [138]: # <!-- Student -->
          logicArray = arrayUp > 3
          newArrayUp = arrayUp[logicArray]
          print("logicArray",logicArray)
          print("newArrayUp",newArrayUp)
```

```
logicArray [False False False False  True  True]
newArrayUp [4. 5.]
```

The above lines can usefully be combined.

```
In [139]: # <!-- Student -->
          newArrayDown = arrayDown[arrayDown < 2]
          print("newArrayDown",newArrayDown)
```

```
newArrayDown [1. 0.]
```

9.8 Writing and reading numpy arrays

Generate some Monte Carlo data (see Section 12) and write it out as a csv file.

```
In [140]: # <!-- Student -->
          nEvents = 100
          mean = 6.0
          RMS = 2.0
          normDistArr = np.random.normal(loc = mean, scale = RMS, size = nEvents)
          print("normDistArr\n",normDistArr)
          np.savetxt('normDistArr.csv', normDistArr, delimiter = ',')
```

```
normDistArr
 [ 8.1453  7.6193  7.4936  7.8196  4.3471  6.6485  6.3009  4.9928  3.3227
  5.0513  8.1799  7.0202  7.6854  7.9563  5.7859  6.1293  1.1102  7.934
  6.2786  3.6583  4.6833  6.709   6.9329  5.6155  3.9292  3.5441  8.9621
  3.6516  4.6672  7.198   5.9026  7.9709  5.3689  5.1028  4.2562  7.948
  7.6267  3.7364  4.6793  6.3398  7.105   2.8039  5.0949  8.1553  4.1589
  3.6227  6.8131  5.0224  8.3582  6.2436  9.2278  6.1067  5.6475  5.2712
  5.6093  7.4899  6.4517  4.1713  6.6965  6.8049  0.8176  5.4163  2.0443
  1.2363  4.1542  4.4677  7.796   5.283   5.3309  4.3617  3.5955  8.4288
  3.6288  4.4145 10.5566  8.5931  1.3051  6.5533  4.5855  2.3502  5.9195
  6.5625  5.2512  4.6367  5.3097  6.6777  9.5263  6.183   3.5901  4.1161
  6.293   4.0872  4.5936  6.9117  8.2514  9.5149  4.6873  4.5017  3.1084
  5.8599]
```

```
In [141]: # <!-- Student -->
          gaussArr = np.loadtxt("normDistArr.csv")
          print("gaussArr\n",gaussArr)
```

```
gaussArr
 [ 8.1453  7.6193  7.4936  7.8196  4.3471  6.6485  6.3009  4.9928  3.3227
  5.0513  8.1799  7.0202  7.6854  7.9563  5.7859  6.1293  1.1102  7.934
  6.2786  3.6583  4.6833  6.709   6.9329  5.6155  3.9292  3.5441  8.9621
  3.6516  4.6672  7.198   5.9026  7.9709  5.3689  5.1028  4.2562  7.948
  7.6267  3.7364  4.6793  6.3398  7.105   2.8039  5.0949  8.1553  4.1589
  3.6227  6.8131  5.0224  8.3582  6.2436  9.2278  6.1067  5.6475  5.2712
  5.6093  7.4899  6.4517  4.1713  6.6965  6.8049  0.8176  5.4163  2.0443
  1.2363  4.1542  4.4677  7.796   5.283   5.3309  4.3617  3.5955  8.4288
  3.6288  4.4145 10.5566  8.5931  1.3051  6.5533  4.5855  2.3502  5.9195
  6.5625  5.2512  4.6367  5.3097  6.6777  9.5263  6.183   3.5901  4.1161
  6.293   4.0872  4.5936  6.9117  8.2514  9.5149  4.6873  4.5017  3.1084
  5.8599]
```

9.9 Type conversion for numpy arrays

The type of arrays can be changed as follows.

```
In [142]: # <!-- Student -->
          #
          fData = np.linspace(0.1, 17.3, 5)
          iData1 = np.int_(fData)
          iData2 = fData.astype(int)
          print("fData",fData)
          print("iData1",iData1)
          print("iData2",iData2)
```

```
fData [ 0.1  4.4  8.7 13.  17.3]
iData1 [ 0  4  8 12 17]
iData2 [ 0  4  8 12 17]
```

```
In [143]: # <!-- Student -->
          #
          bData = np.ones(4).astype(bool)
          fData1 = bData.astype(float)
          iData = bData.astype(int)
          print("bData",bData)
          print("fData1",fData1)
          print("iData",iData)
```

```
bData [ True  True  True  True]
fData1 [1. 1. 1. 1.]
iData [1 1 1 1]
```

9.10 Printing numpy arrays

The precision with which arrays are printed can be steered.

```
In [144]: # <!-- Student -->
          np.set_printoptions(precision = 1)
          print(gaussArr)
```

```
[ 8.1  7.6  7.5  7.8  4.3  6.6  6.3  5.   3.3  5.1  8.2  7.   7.7  8.
  5.8  6.1  1.1  7.9  6.3  3.7  4.7  6.7  6.9  5.6  3.9  3.5  9.   3.7
  4.7  7.2  5.9  8.   5.4  5.1  4.3  7.9  7.6  3.7  4.7  6.3  7.1  2.8
  5.1  8.2  4.2  3.6  6.8  5.   8.4  6.2  9.2  6.1  5.6  5.3  5.6  7.5
  6.5  4.2  6.7  6.8  0.8  5.4  2.   1.2  4.2  4.5  7.8  5.3  5.3  4.4
  3.6  8.4  3.6  4.4 10.6  8.6  1.3  6.6  4.6  2.4  5.9  6.6  5.3  4.6
  5.3  6.7  9.5  6.2  3.6  4.1  6.3  4.1  4.6  6.9  8.3  9.5  4.7  4.5
  3.1  5.9]
```

10 Python control structures

10.1 For loop

```
In [145]: # <!-- Student -->
          start = 0
          stop = 5
          step = 2
          #
          for index in range(start, stop, step):
              print("Index",index)
          print("Final value of index",index)
          print(" ")
```

```
Index 0
Index 2
Index 4
Final value of index 4
```

```
In [146]: # <!-- Student -->
          start = 0
          stop = 3
          for index in range(start, stop):
              print("Index",index)
          print("Final value of index",index)
```

```
Index 0
Index 1
Index 2
Final value of index 2
```

For loops can also be used to cycle through the elements in a list or tuple.

```
In [147]: # <!-- Student -->
          loopList = ["one", "two", "three"]
          for var in loopList:
              print(var)
          print("End of loop, var is",var)
```

```
one
two
three
End of loop, var is three
```

```
In [148]: # <!-- Student -->
          loopTuple = ("A", "B", "C")
          print(" ")
          for var in loopTuple:
              print(var)
          print("End of loop, var is",var)
```

```
A
B
C
End of loop, var is C
```

10.2 While statement

```
In [149]: # <!-- Student -->
          #
          test = 0.3
          limit = 1.1
          step = 0.25
          while test < limit:
              print("test =",test)
              test = test + step
          print("Final value of test is",test)
```

```
test = 0.3
test = 0.55
test = 0.8
test = 1.05
Final value of test is 1.3
```

10.3 If statement

```
In [150]: # <!-- Student -->
          test = 2.3
          if test < 1.0:
              print("This is section A")
          elif test > 2.0 and test < 3.0:
              print("This is section B")
          elif test > 3.0 and test < 4.0:
              print("This is section C")
          else:
              print("This is section D")
          print("This is the end of the if statement, the value of test is",test)
```

```
This is section B
This is the end of the if statement, the value of test is 2.3
```

10.4 Continue and break

```
In [151]: # <!-- Student -->
          for letter in "Constantinople":
              if letter in "a, e, i, o, u":
                  continue
              print(letter)
          print("Final value of letter is",letter)
```

```
C
n
s
t
n
t
n
p
l
Final value of letter is e
```

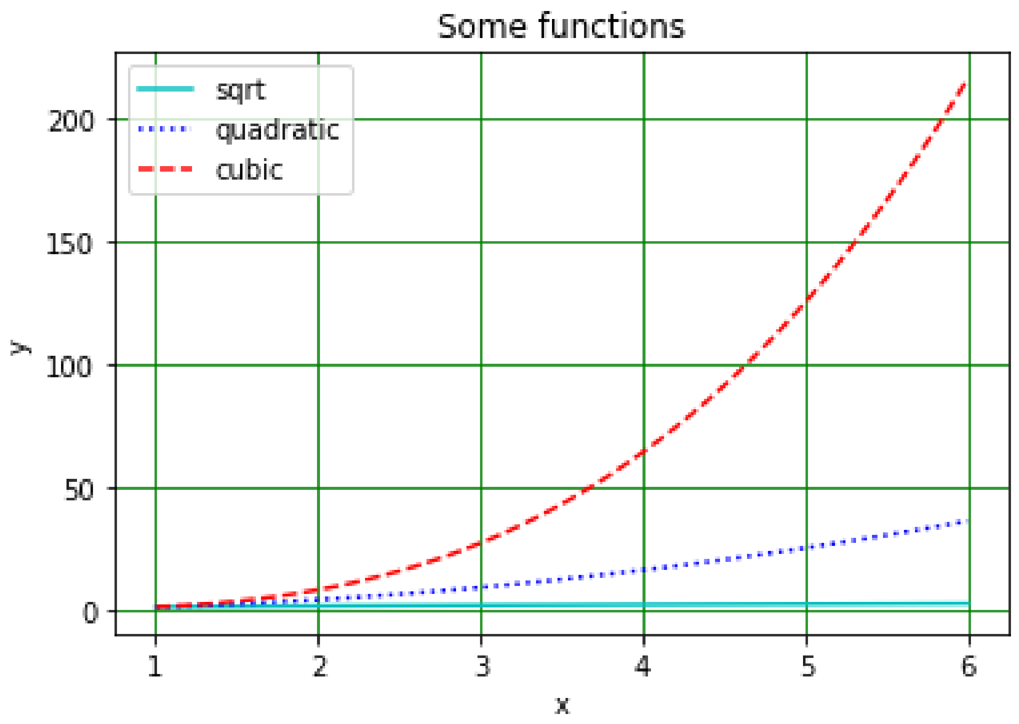
```
In [152]: # <!-- Student -->
          for letter in "Constantinople":
              if letter in "a, e, i, o, u":
                  break
              print(letter)
          print("Final value of letter is",letter)
```

```
C
Final value of letter is o
```

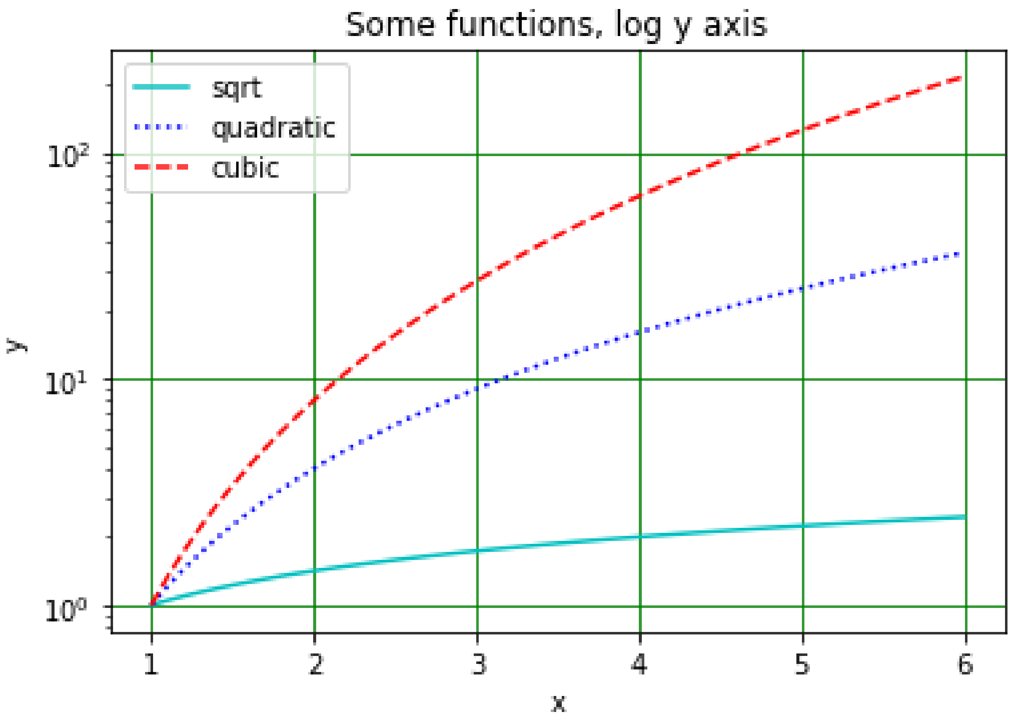
11 Plotting

11.1 Lines

```
In [153]: xArr = np.linspace(1, 6, 51)
          yArr05 = np.sqrt(xArr)
          yArr2 = xArr**2
          yArr3 = xArr**3
          plt.figure(figsize = (6, 4))
          plt.title("Some functions")
          plt.xlabel("x")
          plt.ylabel("y")
          plt.plot(xArr, yArr05, linestyle = '-', color = 'c', label = "sqrt")
          plt.plot(xArr, yArr2, linestyle = ':', color = 'b', label = "quadratic")
          plt.plot(xArr, yArr3, linestyle = '--', color = 'r', label = "cubic")
          plt.legend()
          plt.grid(color = 'g')
```



```
In [154]: plt.figure(figsize = (6, 4))
          plt.title("Some functions, log y axis")
          plt.xlabel("x")
          plt.ylabel("y")
          plt.plot(xArr, yArr05, linestyle = '-', color = 'c', label = "sqrt")
          plt.plot(xArr, yArr2, linestyle = ':', color = 'b', label = "quadratic")
          plt.plot(xArr, yArr3, linestyle = '--', color = 'r', label = "cubic")
          plt.yscale('log')
          plt.legend()
          plt.grid(color = 'g')
```

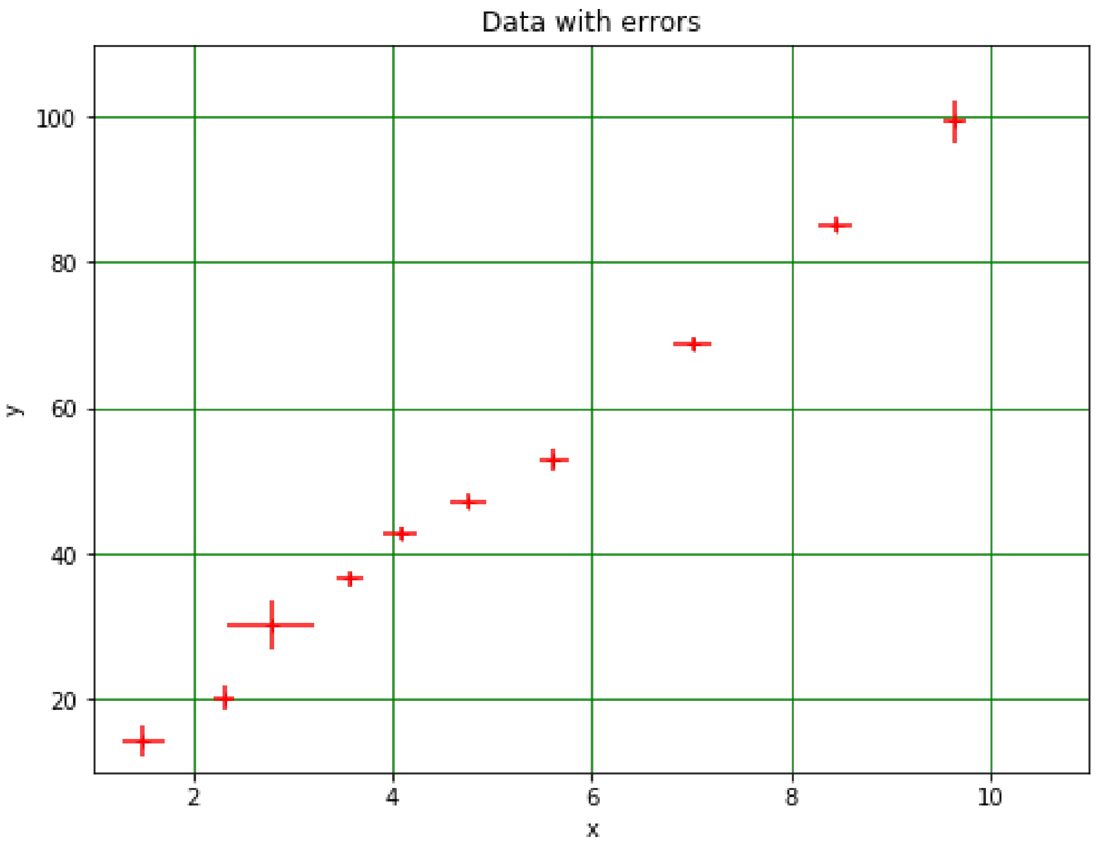


11.2 Points with errorbars

```
In [155]: # <!-- Student -->
          # nPoints is the number of data points
          nPoints = 10
          #
          # Define numpy arrays, initially filled with zeros, to store x and y values
          xData = np.zeros(nPoints)
          yData = np.zeros(nPoints)
          #
          # Define arrays to store the errors in x and y
          xError = np.zeros(nPoints)
          yError = np.zeros(nPoints)
          #
          # Enter the data
          xData[0] = 1.50
          xData[1] = 2.31
          xData[2] = 2.78
          xData[3] = 3.58
          xData[4] = 4.08
          xData[5] = 4.76
          xData[6] = 5.62
          xData[7] = 7.02
```

```
xData[8] = 8.45
xData[9] = 9.65
#
yData[0] = 14.3
yData[1] = 20.2
yData[2] = 30.1
yData[3] = 36.5
yData[4] = 42.7
yData[5] = 47.1
yData[6] = 52.9
yData[7] = 68.8
yData[8] = 85.2
yData[9] = 99.4
#
# Enter the errors
xError[0] = 0.21
xError[1] = 0.11
xError[2] = 0.43
xError[3] = 0.13
xError[4] = 0.17
xError[5] = 0.18
xError[6] = 0.15
xError[7] = 0.19
xError[8] = 0.17
xError[9] = 0.11
#
yError[0] = 2.1
yError[1] = 1.7
yError[2] = 3.3
yError[3] = 1.1
yError[4] = 0.9
yError[5] = 1.1
yError[6] = 1.5
yError[7] = 0.9
yError[8] = 1.2
yError[9] = 2.9
#
# Plot data
fig = plt.figure(figsize = (8, 6))
plt.title('Data with errors')
plt.xlabel('x')
plt.ylabel('y')
plt.errorbar(xData, yData, xerr = xError, yerr = yError, color = 'r',
             marker = '+', linestyle = '', label = "Data")
plt.xlim(1.0, 11.0)
plt.ylim(10.0, 110.0)
plt.grid(color = 'g')
plt.savefig("ErrorBarPlot.png")
```

```
        plt.show()
```



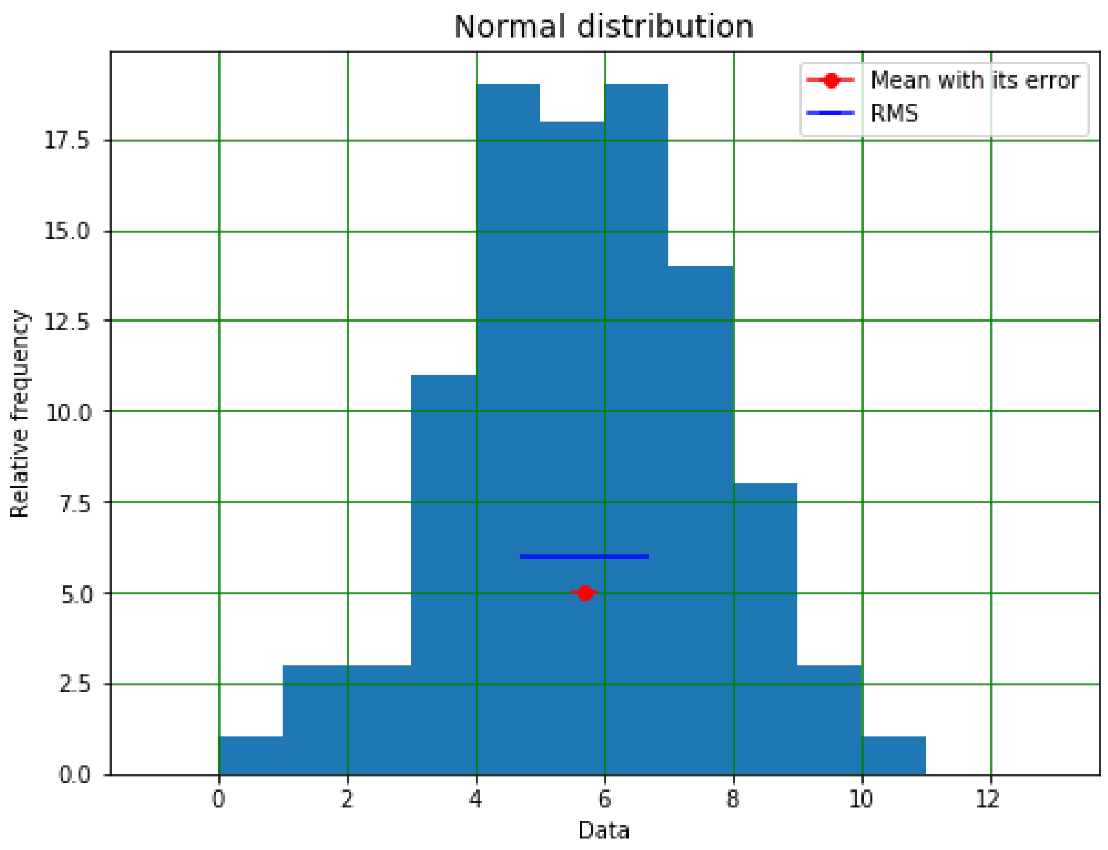
11.3 Histograms (with mean and standard deviation)

A histogram shows the frequency with which values occur in a dataset. The data are split into "bins", the lower and upper limits of which are indicated by the edges of the bars in the plot. The area of each of the bars is proportional to the number of data points that fall within the bin. (Most histograms use bins of equal width, in which case the height of the bin indicates the number of entries it contains.) Here, a histogram is plotted using defined, rather than accepting those calculated by matplotlib. (A complete description of `plt.hist` is provided here.)

The mean and standard deviation of the distribution are also calculated and displayed on the plot.

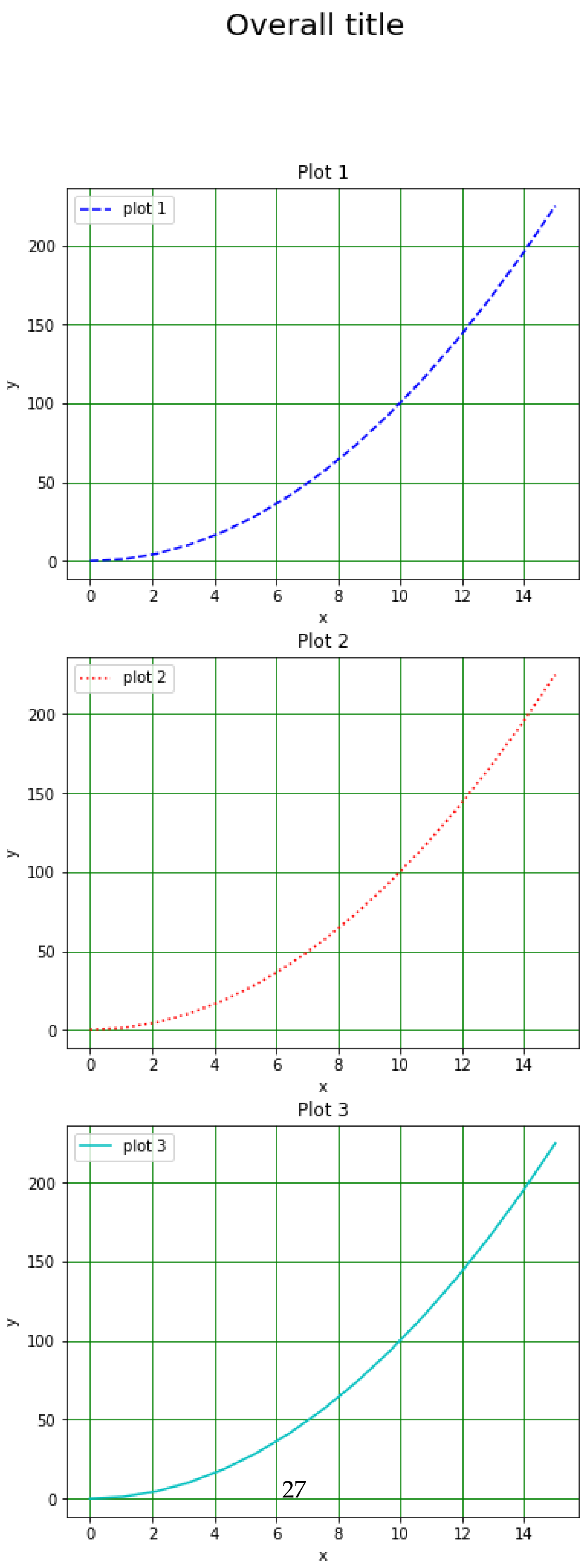
```
In [156]: # <!-- Student -->
          #
          binBot = -1.0
          binTop = 13.0
          binNumber = 14
          binEdges = np.linspace(binBot, binTop, binNumber + 1)
          binWidth = (binTop - binBot)/binNumber
          print("Histogram bins start at",binBot,"finish at",binTop)
          print("Number of bins is",binNumber,"and width of bins is",binWidth)
          #
          nEvents = len(gaussArr) # determine length of gaussArr
          mu = np.mean(gaussArr) # calculate arithmetic mean of numbers in array
          sigma = np.std(gaussArr) # calculate standard deviation (error on single value)
          muError = sigma/np.sqrt(nEvents) # calculate error of mean
          yMu = nEvents/20
          ySigma = 1.2*nEvents/20
          #
          plt.figure(figsize = (8, 6))
          plt.title('Normal distribution', fontsize = 14)
          plt.xlabel('Data')
          plt.ylabel('Relative frequency')
          plt.hist(gaussArr, bins = binEdges)
          plt.errorbar(mu, yMu, xerr = muError, marker = 'o', color = 'r',
                       label = 'Mean with its error')
          plt.errorbar(mu, ySigma, xerr = sigma/2, marker = '', color = 'b', label = 'RMS')
          plt.grid(color = 'g')
          plt.legend()
          plt.show()
```

```
Histogram bins start at -1.0 finish at 13.0
Number of bins is 14 and width of bins is 1.0
```



11.4 Multiple plots in one figure

```
In [157]: # <!-- Student -->
          #
          xArr = np.linspace(0, 15, 15)
          yArr = xArr**2
          #
          fig = plt.figure(figsize = (6, 16)) # opens a figure
          fig.suptitle('Overall title', fontsize=20) # overall title
          #
          plt.subplot(3, 1, 1) # creates 3 row, 1 column grid, starts in top left square
          plt.title("Plot 1")
          plt.xlabel("x")
          plt.ylabel("y")
          plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'plot 1')
          plt.legend()
          plt.grid(color = 'g')
          #
          plt.subplot(3, 1, 2) # second square, reading from left to right, top to bottom
          plt.title("Plot 2")
          plt.xlabel("x")
          plt.ylabel("y")
          plt.plot(xArr, yArr, linestyle = ':', color = 'r', label = 'plot 2')
          plt.grid(color = 'g')
          plt.legend()
          #
          plt.subplot(3, 1, 3) # plot in third square
          plt.title("Plot 3")
          plt.xlabel("x")
          plt.ylabel("y")
          plt.plot(xArr, yArr, linestyle = '-', color = 'c', label = 'plot 3')
          plt.grid(color = 'g')
          plt.legend()
          #
          plt.show()
```


Overall title
Plot 1
plot 1
Plot 2
plot 2
Plot 3
plot 3
x
y
0
2
4
6
8
10
12
14
50
100
150
200

12 Generating random numbers using numpy

```
In [158]: # <!-- Student -->
          import numpy as np
          #
          # Set seed if want to generate same random numbers, otherwise omit!
          np.random.seed(1327)
          #
          # generate one random number
          x = np.random.rand()
          print("x = ",x)
          #
          # generate an array of 4 random numbers
          x1D = np.random.rand(4)
          print(" ")
          print("x1D = \n",x1D)
          #
          # generate a 2D array containing 3 rows and 2 columns of random numbers
          x2D = np.random.rand(3, 2)
          print(" ")
          print("x2D = \n",x2D)
```

```
x =  0.5718839411688054

x1D =
 [0.6 0.1 0.3 0.2]

x2D =
 [[0.1 0.5]
 [0.9 0.9]
 [0.3 0.8]]
```

12.1 Poisson distribution

```
In [159]: # <!-- Student -->
          #
          lam = 3.2
          nEvents = 10
          randArr = np.random.poisson(lam, nEvents)
          print("randArr\n",randArr[0:10])
```

```
randArr
 [3 1 4 3 3 2 8 3 1 1]
```

12.2 Normal distribution

```
In [160]: # <!-- Student -->
          #
          mean = 3.2
          RMS = 2.1
          nEvents = 10
          randArr = np.random.normal(mean, RMS, nEvents)
          np.set_printoptions(precision = 4)
          print("randArr\n",randArr[0:5])
```

```
randArr
 [4.4446 1.3705 2.6073 2.4468 3.3691]
```

13 Symbolic programming

13.1 Defining sympy equations

```
In [161]: # <!-- Student -->
          #
          x, a, b, c = sp.symbols('x a b c')
          quadEq =  sp.Eq(a*x**2 + b*x + c, 0)
          print("The quadratic equation is",quadEq)
```

```
The quadratic equation is Eq(a*x**2 + b*x + c, 0)
```

13.2 Solving equations

```
In [162]: # <!-- Student -->
          #
          import sympy as sp
          quadSol = sp.solve(quadEq, x)
          print("Solution of",quadEq,"is x =\n",quadSol)
```

```
Solution of Eq(a*x**2 + b*x + c, 0) is x =
 [(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]
```

```
In [163]: # <!-- Student -->
          #
          nQuadSols = len(quadSol)
          for n in range(0, nQuadSols):
              print("Solution",n,"is",quadSol[n])
```

```
Solution 0 is (-b + sqrt(-4*a*c + b**2))/(2*a)
Solution 1 is -(b + sqrt(-4*a*c + b**2))/(2*a)
```

Note that in order to use `sp.solve`, the RHS of the equation has to be zero.

13.3 Defining a symbolic function

```
In [164]: # <!-- Student -->
          #
          def quad(x):
              f = a*x**2 + b*x + c
              return f
          #
          print("Solution of quad(x) = 0 is x =\n",sp.solve(quad(x), x))
```

```
Solution of quad(x) = 0 is x =
 [(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]
```

Evaluate `quad(x)` for a specific value of `x`:

```
In [165]: # <!-- Student -->
          #
          print("quad(2) =",quad(2))
```

```
quad(2) = 4*a + 2*b + c
```

Substitute the numerical value 3 for `a` using `subs(a, 3)`:

```
In [166]: # <!-- Student -->
          print("quad(x).subs(a, 3) =",quad(x).subs(a, 3))
```

```
quad(x).subs(a, 3) = b*x + c + 3*x**2
```

We can substitute more than one value into an expression:

```
In [167]: # <!-- Student -->
          #
          print("quad(x).subs(a, 3).subs(b = -2.4).subs(c = 17/3) =\n",
                quad(x).subs(a, 3).subs(b, -2.4).subs(c, 17/3))
```

```
quad(x).subs(a, 3).subs(b = -2.4).subs(c = 17/3) =
 3*x**2 - 2.4*x + 5.66666666666667
```

We can also substitute another symbolic value:

```
In [168]: # <!-- Student -->
          #
          print("quad(x).subs(a, x) =",quad(x).subs(a, x))
```

```
quad(x).subs(a, x) = b*x + c + x**3
```

13.4 Solveset

```
In [169]: # <!-- Student -->
          #
          z, p, q, r = sp.symbols('z p q r')
          newQuadEq =  sp.Eq(p*z**2 + q*z, -r)
          print("This quadratic equation is",newQuadEq)
          #
          newQuadSol = sp.solveset(newQuadEq, z)
          print("Its solution is\n",newQuadSol)
```

```
This quadratic equation is Eq(p*z**2 + q*z, -r)
Its solution is
 {-q/(2*p) - sqrt(-4*p*r + q**2)/(2*p), -q/(2*p) + sqrt(-4*p*r + q**2)/(2*p)}
```

The sp.solveset solutions are a set. This allows sp.solveset to deal with situations where there is an infinite number of solutions, but means these have to be accessed in a slightly different way. If the number of solutions is finite, the following works.

```
In [170]: # <!-- Student -->
          #
          nSol = 0
          for sol in newQuadSol:
              print("Solution",nSol,"is",sol)
```

```
Solution 0 is -q/(2*p) - sqrt(-4*p*r + q**2)/(2*p)
Solution 0 is -q/(2*p) + sqrt(-4*p*r + q**2)/(2*p)
```

13.5 Fractions

```
In [171]: # <!-- Student -->
          #
          alpha = sp.Rational(1, 137)
          print("alpha =",alpha)
```

```
alpha = 1/137
```

They can be added, subtracted, multiplied and divided.

```
In [172]: # <!-- Student -->
          #
          half = sp.Rational(1, 2)
          quarter = sp.Rational(1, 4)
          print("half + quarter =",half + quarter)
          print("half - quarter =",half - quarter)
          print("half*quarter =",half*quarter)
          print("half/quarter =",half/quarter)
```

```
half + quarter = 3/4
half - quarter = 1/4
half*quarter = 1/8
half/quarter = 2
```

13.6 Differentiation

```
In [173]: # <!-- Student -->
          #
          print("The differential of a*x**5 + b*x**2 + c/x**3 w.r.t. x is\n",
                sp.diff(a*x**5 + b*x**2 + c/x**3, x))
```

```
The differential of a*x**5 + b*x**2 + c/x**3 w.r.t. x is
 5*a*x**4 + 2*b*x - 3*c/x**4
```

The second (and higher) differentials can also be calculated (in several ways).

```
In [174]: # <!-- Student -->
          #
          print("Second differential is",
                sp.diff(sp.diff(a*x**5 + b*x**2 + c/x**3, x),x))
          print("Second differential is",
                sp.diff(a*x**5 + b*x**2 + c/x**3, x, x))
          print("Second differential is",
                sp.diff(a*x**5 + b*x**2 + c/x**3, x, 2))
```

```
Second differential is 20*a*x**3 + 2*b + 12*c/x**5
Second differential is 2*(10*a*x**3 + b + 6*c/x**5)
Second differential is 2*(10*a*x**3 + b + 6*c/x**5)
```

Notice the output from the first of these methods is written differently to that from the latter two, but we can check they are the same by subtracting them and using `sp.simplify` to simplify the result!

```
In [175]: #<!-- Student -->
          #
          sp.simplify(sp.diff(sp.diff(a*x**5 + b*x**2 + c/x**3, x),x)
                      - sp.diff(a*x**5 + b*x**2 + c/x**3, x, x))
```

Out[175]:

$$0$$

13.7 Integration

Indefinite integrals:

```
In [176]: # <!-- Student -->
          #
          intQuad = sp.integrate(quad(x), x)
          print("Integral of",quad(x),"w.r.t. x is",intQuad)
```

```
Integral of a*x**2 + b*x + c w.r.t. x is a*x**3/3 + b*x**2/2 + c*x
```

Evaluate with particular values of a, b, c and x using `sp.subs`.

```
In [177]: # <!-- Student -->
          #
          print("Integral of",quad(x),"with a = 1, b = 2, c = -1 and x = 4 is",
                intQuad.subs(a, 1).subs(b, 2).subs(c, -1).subs(x, 4))
```

```
Integral of a*x**2 + b*x + c with a = 1, b = 2, c = -1 and x = 4 is 100/3
```

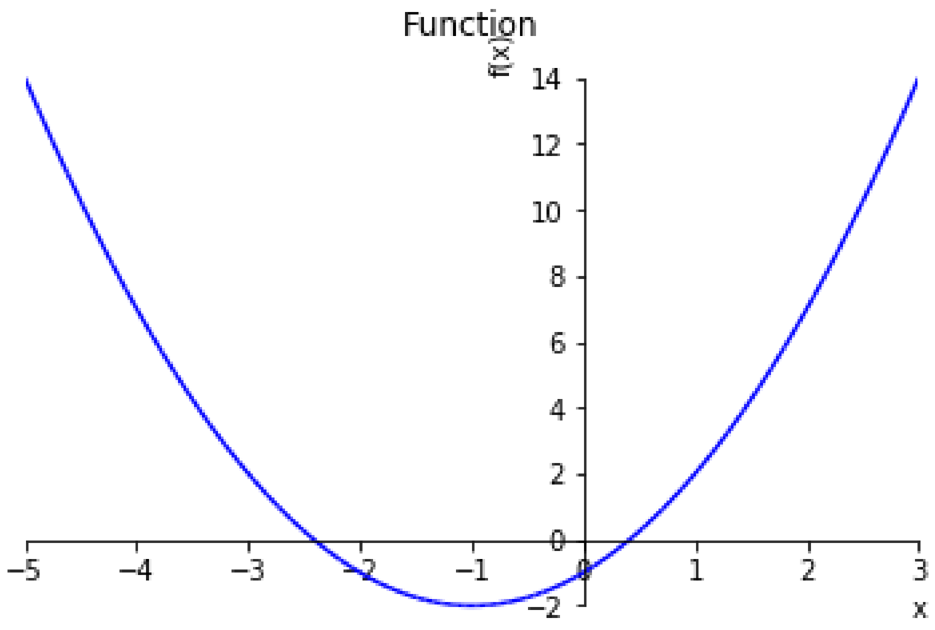
Definite integrals.

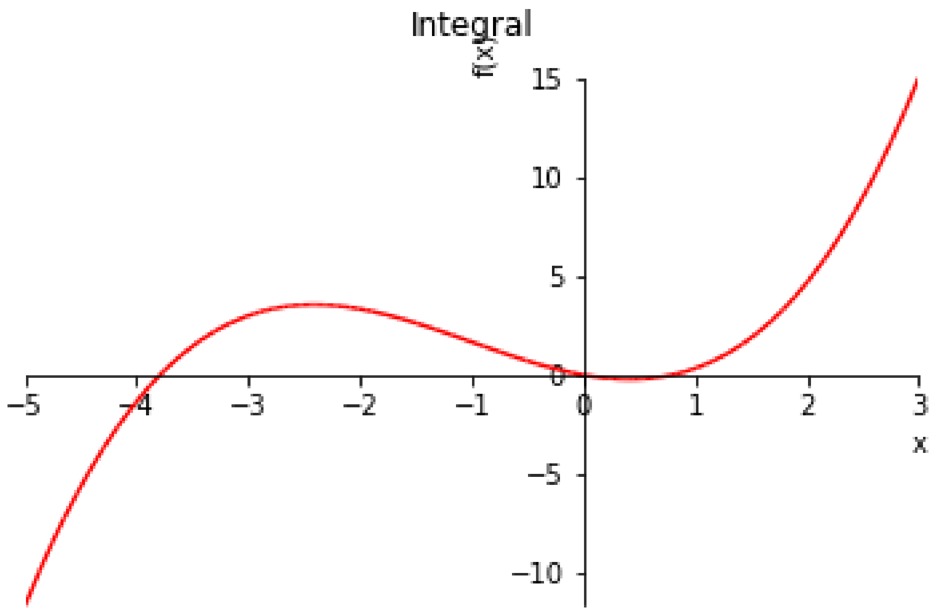
```
In [178]: # <!-- Student -->
          #
          intQuadLim = sp.integrate(quad(x), (x, -1, 2))
          print("Integral of",quad(x),"w.r.t. x over range -1 to 2 is",intQuadLim)
```

```
Integral of a*x**2 + b*x + c w.r.t. x over range -1 to 2 is 3*a + 3*b/2 + 3*c
```

13.8 Plotting with sympy

```
In [179]: # <!-- Student -->
          #
          import matplotlib.pyplot as plt
          %matplotlib inline
          #
          sp.plot(quad(x).subs(a, 1).subs(b, 2).subs(c, -1),(x, -5, 3),
                  line_color = 'blue', title = "Function")
          sp.plot(intQuad.subs(a, 1).subs(b, 2).subs(c, -1).subs(x, x),(x, -5, 3),
                  line_color = 'red', title = "Integral")
          plt.show()
```

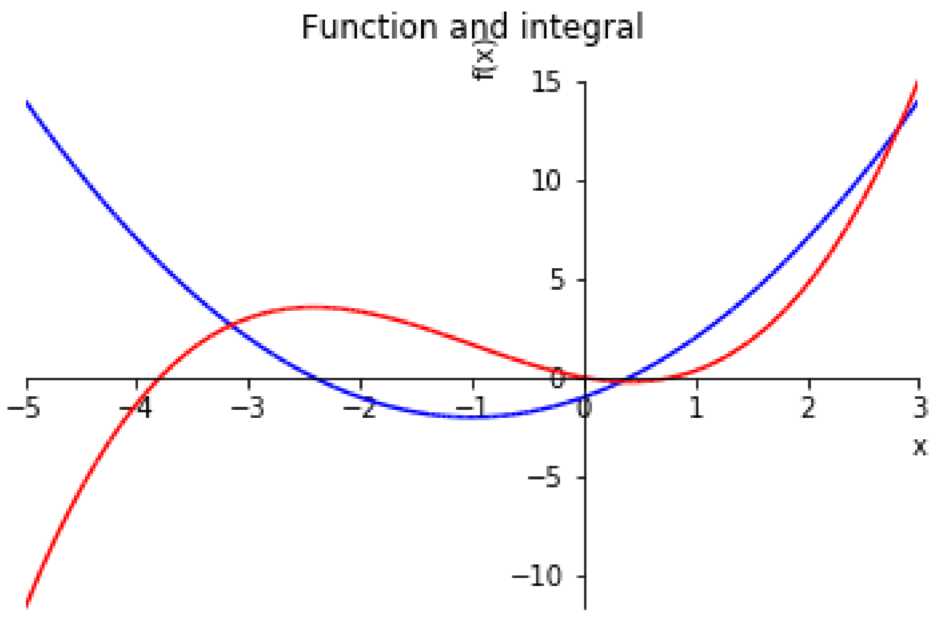




Two plots in one figure with different line colours:

```
In [180]: # <!-- Student -->
          #
```

```
thisPlot = sp.plot(quad(x).subs(a, 1).subs(b, 2).subs(c, -1),
                   intQuad.subs(a, 1).subs(b, 2).subs(c, -1).subs(x, x),
                   (x, -5, 3), show = False, title = "Function and integral")
thisPlot[0].line_color = 'blue'
thisPlot[1].line_color = 'red'
thisPlot.show()
```



13.9 Solving differential equations

```
In [181]: # <!-- Student -->
          #
          t = sp.symbols("t")
          g = sp.Function("g")
          diffEq = sp.Eq(g(t).diff(t), sp.exp(-t) - g(t))
          solDiffEq = sp.dsolve(diffEq, g(t))
          print("Solution of differential equation is",solDiffEq)
```

```
Solution of differential equation is Eq(g(t), (C1 + t)*exp(-t))
```

Alternative method of sloving the equation requires that we rewrite it so that the RHS is zero: We can then solve it as below. Note also alternative way of defining $\frac{d}{dt}g(t)$.

```
In [182]: # <!-- Student -->
          #
          g_ = sp.Derivative(g(t), t)
```

```
         print("Solution of differential equation is",
               sp.dsolve(g_ - sp.exp(-t) + g(t), g(t)))
```

```
Solution of differential equation is Eq(g(t), (C1 + t)*exp(-t))
```

Find the arbitrary constant C1.

```
In [183]: # <!-- Student -->
          #
          equationForC1 = solDiffEq.subs([(t, 0), (g(0), 1)])
          print("Equation for C1 is",equationForC1)
```

```
Equation for C1 is Eq(1, C1)
```

13.10 Series

To use Taylor series must specify the function we want to express as a series and the variable in which we want the expansion to be done. We can also specify the point around which the expansion should be made ($\pi/4$ in the example below) and the maximum number of terms (3 below). Look at the documentation for more information.

```
In [184]: # <!-- Student -->
          sinSeries = sp.sin(x).series(x, sp.pi/4, 2)
          print("First two terms in Taylor series for sine function about pi/4 are\n",
                sinSeries)
```

```
First two terms in Taylor series for sine function about pi/4 are
 sqrt(2)/2 + sqrt(2)*(x - pi/4)/2 + O((x - pi/4)**2, (x, pi/4))
```

13.11 Matrices and linear equations

Matrices can be defined and inverted as shown here.

```
In [185]: # <!-- Student -->
          #
          A = sp.Matrix(([3, 7], [4, -2]))
          invA = A.inv()
          AinvA = A*invA
          invAA = invA*A
          print("This is the matrix A\n",A)
          print("Its inverse is\n",invA)
          print("A multplied by its inverse is\n",AinvA)
          print("The inverse of A multplied y A is\n",invAA)
          print("As expected, both multiplications give the unit matrix!")
```

```
This is the matrix A
 Matrix([[3, 7], [4, -2]])
Its inverse is
 Matrix([[1/17, 7/34], [2/17, -3/34]])
A multplied by its inverse is
 Matrix([[1, 0], [0, 1]])
The inverse of A multplied y A is
 Matrix([[1, 0], [0, 1]])
As expected, both multiplications give the unit matrix!
```

Systems of linear equations can be defined and solved using matrices. For example, consider the equations:

$$\begin{aligned} 3x + 4y &= 7 \\ x - 3y &= 2. \end{aligned}$$

```
In [186]: # <!-- Student -->
          #
          x, y = sp.symbols('x y')
          M = sp.Matrix(([3, 4], [1, -3]))
          X = sp.Matrix((x, y))
          C = sp.Matrix((7, 2))
          invM = M.inv()
          solVect = invM*C
          print("Equations to solve",M*X,'=',C)
          print("Solution is",X,"=",solVect)
```

```
Equations to solve Matrix([[3*x + 4*y], [x - 3*y]]) = Matrix([[7], [2]])
Solution is Matrix([[x], [y]]) = Matrix([[29/13], [1/13]])
```

13.12 Printing LaTeX source from sympy

You can print equations from sympy in LaTeX format, so they can be copied and pasted into Markdown cells or documents. An example is shown below.

```
In [187]: # <!-- Student -->
          #
          print(sp.latex(sp.Integral(sp.sqrt(1/x), x)))
```

```
\int \sqrt{\frac{1}{x}}\, dx
```

13.13 Writing LaTeX equations to screen from sympy

```
In [ ]: # <!-- Student -->
        #
        sp.init_printing()
        #
        sp.solve(a*x**2 + b*x + c, x)
```

14 Comments on good coding practice

1) Use variable names that are self-explanatory and clearly different. It's not a good idea to use `X` and `x` as labels for arrays containing the measurements of the period of a pendulum and its mass, for example. Names like periodArr and massArr are much clearer. (I find it often helps to indicate whether a variable is a single value or an array, e.g. `t` could be a single time and `tArr` an array containin lots of times.) Good variable names can save you having to write lots of comments.

2) If you are using the same quantity two or more times, don't give it a new name each time. That will just be confusing. For example, if you have an array of ten mass values, used three times, don't call it `massArray` the first time, `massArr` the second and `tenMass` the third!

3) Put sensible comments in your code (i.e. explain the things that need to be explained but not the things that don't!). For example, this is a useless comment:

```
#
# Print out the value of y
print("Value of y is",y)
```

Make sure you explain what your code is supposed to do in a concise way. Often the nitty-gritty of things like how a numpy routine works are best left to the official documentation, particularly if the name makes it clear what the routine is doing!

```
#
nRow = 0
nonZero = 7
rowTest = 13
shortData = np.zeros((rowTest + 2, nonZero + 2))
shortData[3, :] = 1.0
#
# Find row in shortData that contains more than nonZero elements which are not 0.
while nRow < rowTest:
    if np.count_nonzero(shortData, axis = 1)[nRow] > nonZero:
        print("nRow =",nRow)
        break
    nRow = nRow + 1
```

4) It's a good idea to add units to your code. This can be done with a comment behind the quantity in question.

```
carSpeed = 33.2 # km per hour
hairThickness = 100 # um
```

(Note that um is a frequently used substitute for μm.)

5) Use variables rather than numbers to control the length of arrays, the number of iterations in loops and so on. If you define an array to be of length 10, and use `for` loops with upper limit 10, you'll have to change all the 10s to 12s by hand if you later need a bigger array. If you had used a variable, you would just have had to change 10 to 12 once.

15 Debugging

Examples of some of the error messages Python produces are given in Week 8 of Phys 105. General hints that can help you avoid or track down problesm include:

- Read your code carefully.
- When writing a program, do it in steps and test each step.
- Test your code on examples where you know what the answer should be and check you get what you expect!
- Print out the values of variables during running and check they are as you expect. Do the same for variables after your code has run (the last values will be saved).
- Plot graphs that show your intermediate and final results. Are these sensible?
- Explain your code to someone else. (If there is no-one around, explain it to an imaginary friend!) It's surprising how often this helps you spot a mistake in your logic.
- Look to see if your problem has been encountered before (try google, or look on online forums like stackoverflow.com).
- If you can't solve your problem, try posting it on stackoverflow.com. (Don't do this with Phys105 or Phys106 problems; we don't want to get the volunteers who give up their time trying to solve difficulties too upset!)
- Use the `assert` statement in your code (described below) to check that things you think should be true really are true!

15.1 The assert statement

The assert statement is a debugging tool that is built into Python. It can be used as illustrated in the following example. If the `assert` statment is `False`, the subsequest message is printed and the program terminates.

```
In [189]: # <!-- Student -->
          #
          nQuadEqs = 4
          aArr = np.linspace(1.0, 4.0, nQuadEqs)
          b = 2.0
          c = 3.0
          #
          for n in range(0, nQuadEqs):
              discrim = b**2 - 3*aArr[n]*c
              assert (discrim > 0), "Bumped into negative discrim ({:5.3f}), can't take sqrt!"
```

```
        root1 = (-b + np.sqrt(discrim))/(2*aArr[n])
        root2 = (-b - np.sqrt(discrim))/(2*aArr[n])
        print("For a ={:.2f}, b = {:.2f} and c = {:.2f}, roots are {:.2f} and {:.2f}.".fo
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-189-719c2999b3ef> in <module>
  8 for n in range(0, nQuadEqs):
  9     discrim = b**2 - 3*aArr[n]*c
---> 10     assert (discrim > 0), "Bumped into negative discrim ({:5.3f}), can't take sqrt
 11     root1 = (-b + np.sqrt(discrim))/(2*aArr[n])
 12     root2 = (-b - np.sqrt(discrim))/(2*aArr[n])


AssertionError: Bumped into negative discrim (-5.000), can't take sqrt!
```

16 Creating and using a module

Start by writing a function that we can use to create a module.

```
In [ ]: # <!-- Student -->
        #
        import numpy as np
        #
        def circleParams(r):
            '''
            Given the radius of a circle, this function returns its area and circumference.
            '''
            A = np.pi*r**2
            c = 2*np.pi*r
            return A, c
```

In order to create a module from e.g. this *Python-Summary* Notebook, click on the *File* menu, then on *Download as* and select *Python*. Depending on the security settings on your browser, you may get a warning about the file that is created, saying that it can damage your computer. You can ignore this and click *Keep* or *Save*. In your default download location (usually your *Downloads* folder) you will then have a file called *Python-Summary.py*.

Move *Python-Summary.py* to the directory you are working in and rename it *PythonSummary.py* (no hyphen or other illegal characters!). Open it in your Jupyter Notebook browser. Tidy up the file by deleting the superfluous comment lines - leave the ones that are useful - and other material that isn't part of the required functions. Do not delete lines required for running the functions such as `import numpy as np`.

You can now use all the functions in your file by importing it as a module, as shown in the following example using the function *circleParams* from this Notebook. Note, your file name should have the extension *.py*, but you don't include this in the `import` statement.

Notice that after doing `import PythonSummary as ps`, we have used the routine `ps.circleParams`(cf. using a numpy routine).

```
In [ ]: # <!-- Student -->
        #
        import numpy as np
        import matplotlib.pyplot as plt
        import PythonSummary as ps
        #
        nArr = 50
        rBot = 0.0
        rTop = 4.0
        rArr = np.linspace(rBot, rTop, nArr)
        Aarr, cArr = ps.circleParams(rArr)
        #
        plt.figure(figsize = (7, 5))
        plt.title("Area and circumference of circle")
        plt.ylabel("Area or circumference")
        plt.xlabel("Radius")
        plt.plot(rArr, Aarr, linestyle = '-', color = 'r')
        plt.plot(rArr, cArr, linestyle = '-', color = 'b')
        plt.grid(color = 'green')
        plt.show()
```

We have been careful to include the statement `import numpy as np` at the top of our module. This statement is executed when the module is first loaded, so even if we use our functions from a program which doesn't import numpy, they will work OK.

16.1 Accessing modules in other folders

What we have done so far only allows us to use functions from a module in the directory in which we are working. We can also get at modules in other directories. For example, make a copy of *PythonSummary.py* and call it *PythonSummaryNew.py*. Make a new folder *PythonSummaryLib* in your working directory. (You can do this using *File Explorer* on Windows, *Finder* on a Mac and the command *mkdir* on a Linux system.) Move the file *PythonSummary.py* into the folder *PythonSummaryLib*. Python won't be able to find the *PythonSummaryNew* module until *PythonSummaryLib* has been added to the system variable *path* which lists all the locations searched for files when programs are run. This can be done, and the contents of the *path* variable listed, as follows.

```
In [2]: # <!-- Student -->
        #
        import sys
        #
        sys.path.append('Phys105lib')
        print("Directories in path are:\n",sys.path)
```

```
Directories in path are:
 ['C:\\Users\\green\\OneDrive\\OneDocuments\\Liverpool\\Teaching\\Phys105-2018\\Phys105-All',
```

Once the *path* has been ammended, the command `import PythonSummaryNew as psnew` will function. Note, this addition to *path* will be removed if the kernel, or Jupyter Notebook, is restarted.

The addition to *path* we have made above will allow us to use anything in *PythonSummaryLib* if it is in our current working directory. If we want to be able to use the contents of *PythonSummaryLib* from any directory, we have to add the full description of its location to *path*. On my computer, this implies.

```
In [5]: # <!-- Student -->
        #
        import sys
        #
        sys.path.append('C:/Users/green/OneDrive/OneDocuments/Liverpool/Teaching/Phys105-2018/I
        print("Directories in path are:\n",sys.path)
```

```
Directories in path are:
 ['C:\\Users\\green\\OneDrive\\OneDocuments\\Liverpool\\Teaching\\Phys105-2018\\Phys105-All',
```

You can work out what the full description of the location of *PythonSummaryLib* should be on your computer by looking at the existing entries in your *path* variable.

17 Keyboard input to Python

Python programs can read input from the keyboard. They do this with the function `input()`. The following example shows how it can be used.

```
In [4]: # <!-- Student -->
        #
        name = input("What's your name?")
        print("Nice to meet you " + name + "!")
        age = input("How old are you?")
        print("So, you are already " + str(age) + " years old, " + name + "!")
```

```
What's your name? John


Nice to meet you John!


How old are you? 19


So, you are already 19 years old, John!
```

We see that the input is read as a string, so in the `print` statement, we can concatenate (join together) the strings "Nice to meet you" and `name` using the + operator. This also implies that type conversion will be required if the input is to be used as an integer or a float.