PYTHON THIRD SEMESTER LAB RECORD

Lab record done by Pranav Gopalkrishna, register number 1940223, class 3 CMS

1. Introduction to Python

Python in an a high-level general purpose programming language that is interpreted, rather than compiled. It supports procedural, object-oriented and functional programming. It has a vast array of predefined libraries for numerous functionalities, ranging from formatting output to machine learning.

2. Working with numbers in Python

Here, we learn to carry out basic mathematical operations, define fractions and define complex numbers in Python

2.1. Basic mathematical operations

In [36]:

```
a = 5
b = 7
```

In [37]:

```
print(a + b)
print(a - b)
print(a * b)
print(a / b)
```

```
12
-2
35
0.7142857142857143
```

2.2. Integers

Parsing as integers

To convert a string or float to an integer, we use the int( ) function

In [38]:

```
int(2.3)
```

Out[38]:

```
2
```

In [39]:

```
int("23")
```

Out[39]:

```
23
```

However, a string cannot be a non integer

```
In [40]:
```

```
int("2.3")
```

```
-----------------------------------------------------------------
----------
ValueError                                Traceback (most recent
 call last)
<ipython-input-40-5ed61bf3052f> in <module>()
----> 1 int("2.3")

ValueError: invalid literal for int() with base 10: '2.3'
```

Checking for integers

is_integer( ) is a predefined function defined for float type numbers only

```
In [43]:
```

```
3.4.is_integer()
```

```
Out[43]:

False
```

```
In [44]:
```

```
3.0.is_integer()
```

```
Out[44]:

True
```

The function is not defined for strings

```
In [45]:
```

```
"3.4".is_integer()
```

```
-----------------------------------------------------------------
----------
AttributeError                            Traceback (most recent
 call last)
<ipython-input-45-6b6b18ac6ca9> in <module>()
----> 1 "3.4".is_integer()

AttributeError: 'str' object has no attribute 'is_integer'
```

2.3. Floating point values

A floating point number can be any real number, such as 3, 3.4, 100/33, π, etc.

Parsing as floating point numbers

To convert a string or float to an integer, we use the float( ) function

In [46]:

```
float(3)
```

Out[46]:

```
3.0
```

In [47]:

```
float("3.14")
```

Out[47]:

```
3.14
```

In [48]:

```
float("3")
```

Out[48]:

```
3.0
```

2.4. Fractions

In [56]:

```
from fractions import Fraction
```

Defining fractions

In [57]:

```
frac1 = Fraction(3, 8)
frac2 = Fraction(7, 8)
```

In [58]:

```
print(frac1)
print(frac2)
```

```
3/8
7/8
```

Parsing strings as fractions

To parse a string as a fraction, write a fraction using /, without spaces

For example: 3/5

In [59]:

```
myFraction = "45/67"
```

```
Fraction(myFraction)
```

Out[59]:

```
Fraction(45, 67)
```

Some operations on fractions

Fraction with fraction

In [62]:

```
print(frac1 + frac2)
print(frac1 - frac2)
print(frac1 * frac2)
print(frac1 / frac2)
```

```
5/4
-1/2
21/64
3/7
```

Fraction with any numerical constant

In [63]:

```
x = 4
print(frac1 + x)
print(frac1 - x)
print(frac1 * x)
print(frac1 / x)
```

```
35/8
-29/8
3/2
3/32
```

Here is a program that inputs and operates on two fractions...

In [66]:

```
from fractions import Fraction
def calculate(n1, n2, operator):
    if operator == "+":
        ans = n1 + n2
    elif operator == "-":
        ans = n1 - n2
    elif operator == "*":
        ans = n1 * n2
    elif operator == "/":
        ans = n1 / n2
    else:
        print("Operator not found!")
        return
    print("{0} {1} {2} = {3}".format(n1, operator, n2, ans))
def fractionCalculator():
    try:
        frac1 = Fraction(input("Enter fraction 1: "))
        frac2 = Fraction(input("Enter fraction 2: "))
    except:
        print("Invalid fraction!")
        return
    operator = input("Enter operator: ")
    calculate(frac1, frac2, operator)
fractionCalculator()
```

```
Enter fraction 1: 3/8
Enter fraction 2: 9/7
Enter operator: -
3/8 - 9/7 = -51/56
```

2.5. Complex numbers

Python supports complex numbers with imaginary parts denoted by i or j in the output

Defining complex numbers

In [67]:

```
complex1 = complex(4, 7)
complex2= complex(3, 3)
```

In [68]:

```
print(complex1)
print(complex2)
```

```
(4+7j)
(3+3j)
```

Parsing strings as complex numbers

To parse a string as a complex number, write a complex number without spaces

Note that the real part must come before the imaginary part

Add a 'j' at the end of the imaginary part, since j represents $\sqrt{-1}$

In [69]:

```
myComplexNum = "2+3j"
complex(myComplexNum)
```

Out[69]:

```
(2+3j)
```

Some operations on complex numbers

In [70]:

```
print(complex1 + complex2)
print(complex1 - complex2)
print(complex1 * complex2)
print(complex1 / complex2)
```

```
(7+10j)
(1+4j)


(-9+33j)
(1.8333333333333333+0.5j)
```

NOTE: Modulus (%) and floor division (//) is not valid for complex numbers, as seen below

(Floor division means dividing two numbers and taking the result's floor)

In [71]:

```
complex1 % 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-71-8f2535c34262> in <module>()
----> 1 complex1 % 3

TypeError: can't mod complex numbers.
```

In [72]:

```
complex1 // 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-72-f0906135f151> in <module>()
----> 1 complex1 // 3

TypeError: can't take floor of complex number.
```

Here is a program that inputs and operates on complex numbers...

In [74]:

```
z1 = complex(input("Enter 1st complex number: "))
z2 = complex(input("Enter 2nd complex number: "))
print("Addition of the complex numbers is ", z1+z2)
print("Subtraction of the complex numbers is ", z1-z2)
print("Multiplication of the complex numbers is ", z1*z2)
print("Division of the complex numbers is ", z1/z2)
```

```
Enter 1st complex number: 3+8j
Enter 2nd complex number: 6-9j
Addition of the complex numbers is  (9-1j)
Subtraction of the complex numbers is  (-3+17j)
Multiplication of the complex numbers is  (90+21j)
Division of the complex numbers is  (-0.46153846153846156+0.64102
5641025641j)
```

Isolating real and imaginary parts

In [75]:

```
print(complex1)
print(complex1.real)
print(complex1.imag)
```

```
(4+7j)
4.0
7.0
```

Conjugate

In [76]:

```
print(complex1)
print(complex1.conjugate())
```

```
(4+7j)
(4-7j)
```

Magnitude

Magnitude of a complex number is given by $\sqrt{real^2 + imaginary^2}$

In [77]:

```
print(complex1)
print("The magnitude of the above number is")
print((complex1.real**2 + complex1.imag**2)**0.5)
```

```
(4+7j)
The magnitude of the above number is
8.06225774829855
```

Alternatively, we can use the abs( ) method (absolute value finder) on a complex number to find its magnitude

In [78]:

```
print(complex1)
print("The magnitude of the above number is")
print(abs(complex1))
```

```
(4+7j)
The magnitude of the above number is
8.06225774829855
```

3. Programming structures

3.1. Conditional structures

To check for and act upon certain conditions, we use the if, elif and else statements

Program to check if input is positive or negative

```
In [79]:
```

```python
n = int(input("Enter integer: "))
if n > 0:
    print(n, "is positive.")
elif n < 0:
    print(n, "is negative.")
else:
    print(n, "is zero.")
```

```
Enter integer: -6
-6 is negative.
```

Program to check if input is odd or even

```
In [80]:
```

```python
n = int(input("Enter integer: "))
if n % 2 == 0:
    print(n, "is even.")
else:
    print(n, "is odd.")
```

```
Enter integer: 583
583 is odd.
```

3.2. Loops

Loops allow us to reuse code in consequent repetitions. This is helpful for various tasks, like adding a series, generating a sequence, or printing something repeatedly.

The for loop

The for loop has involves a variable that goes through a range or a set of values. The code under the loop repeats until the variable reaches the last value of the range or set.

```
In [81]:
```

```python
for i in range(1, 6):
    print(i)
```

```
1
2
3
4
5
```

Repeated printing

```
In [82]:
```

```python
for i in range(1, 4):
    print("HA", end = "! ")
```

```
HA! HA! HA!
```

Adding a series

Find the sum of the series $1^2 + 2^2 + 3^2 + \ldots$ for 504 terms

```
In [83]:
```

```python
sum = 0
for i in range(1, 505):
    sum = sum + i*i
print(sum)
```

```
42801780
```

Generating a sequence

Generate 20 terms of the Fibonacci sequence, which starts with $n_1 = 1, n_2 = 1$, and carries on with $n_i = n_{i-1} + n_{i-2}$

```
In [84]:
```

```python
n1 = 0
n2 = 1
for i in range(1, 21):
    print(n2)
    tmp = n2
    n2 = n2 + n1
    n1 = tmp
```

```
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
```

Finding the factors of a number

In [86]:

```
def is_factor(a, b):
    if b % a == 0:
        return True
    else:
        return False
def factors(b):
    for i in range(1, b + 1):
        if is_factor(i, b):
            print(i)
b = input("Enter integer: ")
b = float(b)
if b.is_integer:
    print("Factors...")
    factors(int(b))
```

```
Enter integer: 589
Factors...
1
19
31
589
```

The while loop

The while loop has involves a condition, called the fail condition. If the fail condition is met, the loop terminates. This is a substitute for the for loop, but it can also be used for purposes with non-iterating variables.

User controlled exit

Here is a program that generated how many ever multiples of a number the user wants, which is harder to achieve in a for loop

In [87]:

```
i = 1
n = float(input("Enter a factor: "))
print("{0} X {1} = {2}".format(n, i, n * i))
while True:
    ans = input("Next multiple? (y/n): ")
    if ans == "y":
        i = i + 1
        print("{0} X {1} = {2}".format(n, i, n * i))
    elif ans == "n":
        print("Goodbye.")
        break
    else:
        print("Invalid option!")
```

```
Enter a factor: 4
4.0 X 1 = 4.0
Next multiple? (y/n): y
4.0 X 2 = 8.0
```

```
Next multiple? (y/n): y
4.0 X 3 = 12.0
Next multiple? (y/n): y
4.0 X 4 = 16.0
Next multiple? (y/n): y
4.0 X 5 = 20.0
Next multiple? (y/n): n
Goodbye.
```

4. Lists, tuples and dictionaries

4.1. Lists

Creating a list

In [88]:

```python
myList = [4, 2, 7, 5, 9]
print(myList)
```

```
[4, 2, 7, 5, 9]
```

Empty list

In [89]:

```python
emptyList = []
print(emptyList)
```

```
[]
```

Accessing elements of a list

This involves accessing values in certain indices of the list

Note that a list's indices begin from 0

In [91]:

```python
myList = [4, 2, 7, 5, 9]
i = int(input("Enter index: "))
print(myList[i])
```

```
Enter index: 4
9
```

Iterating through a list's elements

In [92]:

```python
myList = [1, 4, 2, 5, 4, "Hello", "There"]
for i in myList:
    print(i)
```

```
1
4
2
5
4
Hello
There
```

Iterating with indices (using enumerate function)

In [93]:

```
myList = [1, 4, 2, 5, 4, "Hello", "There"]
for index, item in enumerate(myList):
    print(index, item)
```

```
0 1
1 4
2 2
3 5
4 4
5 Hello
6 There
```

Other properties of a list

Also note that a list can be of any data type, and it need not be homogenous in data type

In [94]:

```
myList = ["Meow", 1, 5.4, 'c']
print(myList)
```

```
['Meow', 1, 5.4, 'c']
```

Also note that a list can contain another list or tuple as a single element

In [95]:

```
myList = [[1, 2, 3], ('a', 'b', 'c'), "Hello", "World"]
print(myList)
```

```
[[1, 2, 3], ('a', 'b', 'c'), 'Hello', 'World']
```

Append method

In [96]:

```
myList = []
myList.append("Player")
myList.append(1)
print(myList)
```

```
['Player', 1]
```

Note that argument can also be a list or a tuple...

In [97]:

```
myList.append(["Alpha", "Beta", "Gamma"])
myList.append((1, 2, 3))
myList
```

Out[97]:

```
['Player', 1, ['Alpha', 'Beta', 'Gamma'], (1, 2, 3)]
```

Length of a list

To determine the length of a list i.e. the number of elements in the list, we can use the len( ) function

In [98]:

```
myList = [1, 2, 3, ("I", "J", "K"), "Cat", 3.4]
len(myList)
```

Out[98]:

```
6
```

4.2. Tuples

Creating and handling a tuple is similar to creating a list, but we use parentheses instead of square brackets. Also, append method does not apply for a tuple. While lists are mutable, tuples are immutable.

Creating a tuple

In [99]:

```
myTuple = (1, 2, 3.2 , "Pranav")
print(myTuple)
```

```
(1, 2, 3.2, 'Pranav')
```

Empty tuple

In [100]:

```
emptyTuple = ()
print(emptyTuple)
```

```
()
```

Accessing elements of a tuple

Other than the defintion of the tuple, accessing the elements of a tuple through an index works the exact same way as it does for a list

Iterating through a tuple's elements

Other than the definition of the tuple, iterating through a tuple works the exact same way as iterating through a list

Other properties of a tuple

As with a list, a tuple need have homogenous data types. Also like a list, a tuple can contain other tuples and lists as elements. However, since a tuple is immutable, no new elements can be added, and no elements can be removed.

In [101]:

```
myTuple = ([1, 2, 3], ('a', 'b', 'c'), 2.3 , 3)
print(myTuple)
```

```
([1, 2, 3], ('a', 'b', 'c'), 2.3, 3)
```

Length of a tuple

As with lists, we can use the len( ) function

In [102]:

```
myTuple = ([1, 2, 3], ('a', 'b', 'c'), 2.3 , 3)
len(myTuple)
```

Out[102]:

```
4
```

4.3. Dictionaries

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values. They allow you to index items with your own indices.

Creating a dictionary

To the left of the colon is the index for the element which is to the right of the colon.

In [103]:

```
simpleDictionary = {
    "one" : "I",
    "two" : "II",
    "three" : "III",
    "four" : "IV",
    "five" : "V"
}
print(simpleDictionary)
```

```
{'one': 'I', 'two': 'II', 'three': 'III', 'four': 'IV', 'five':
'V'}
```

Empty dictionary

In [104]:

```python
emptyDictionary = {}
print(emptyDictionary)
```

```
{}
```

Accessing elements of a dictionary

Accessing an element in a dictionary can be done by referring to the index you have given it in the dictionary's definition.

Using square brackets

In [106]:

```python
simpleDictionary = {
    "one" : "I",
    "two" : "II",
    "three" : "III",
    "four" : "IV",
    "five" : "V"
}
i = input("Enter index: ")
simpleDictionary[i]
```

```
Enter index: four
```

Out[106]:

```
'IV'
```

If you enter an invalid index...

In [107]:

```python
simpleDictionary = {
    "one" : "I",
    "two" : "II",
    "three" : "III",
    "four" : "IV",
    "five" : "V"
}
i = input("Enter index: ")
simpleDictionary[i]
```

```
Enter index: 3

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

```
<ipython-input-107-cd9e42f984c1> in <module>()
      7 }
      8 i = input("Enter index: ")
----> 9 simpleDictionary[i]

KeyError: '3'
```

Using the get( ) method

The above result can also be achieved through the get( ) method, as follows. The difference is that if you enter an invalid index in the get( ) function, you will get no output, instead of an error message.

In [108]:

```
simpleDictionary = {
    "one" : "I",
    "two" : "II",
    "three" : "III",
    "four" : "IV",
    "five" : "V"
}
i = input("Enter index: ")
simpleDictionary.get(i)
```

```
Enter index: three
```

Out[108]:

```
'III'
```

If you enter an invalid index...

In [109]:

```
simpleDictionary = {
    "one" : "I",
    "two" : "II",
    "three" : "III",
    "four" : "IV",
    "five" : "V"
}
i = input("Enter index: ")
simpleDictionary.get(i)
```

```
Enter index: 2
```

Iterating through a dictionary's elements

Iterating through values

In [110]:

```
simpleDictionary = {
    "one" : "I",
    "two" : "II",
    "three" : "III",
    "four" : "IV",
    "five" : "V"
}
for i in simpleDictionary.values():
    print(i)
```

```
I
II
III
IV
V
```

Iterating through indices

In [111]:

```
simpleDictionary = {
    "one" : "I",
    "two" : "II",
    "three" : "III",
    "four" : "IV",
    "five" : "V"
}
for i in simpleDictionary:
    print(i)
```

```
one
two
three
four
five
```

Iterating through indices and values together

In [112]:

```
simpleDictionary = {
    "one" : "I",
    "two" : "II",
    "three" : "III",
    "four" : "IV",
    "five" : "V"
}
for i in simpleDictionary.items():
    print(i)
```

```
('one', 'I')
('two', 'II')
('three', 'III')
('four', 'IV')
('five', 'V')
```

Other properties of a dictionary

As with lists and tuples, the index names and elements need not be homogenous in data type

In [113]:

```
mixedUp = {
    3.14 : "Pie",
    "e" : 2.7,
    "Stars" : 0,
    2 : "The Best"
}
for i in mixedUp.items():
    print(i)
```

```
(3.14, 'Pie')
('e', 2.7)

('Stars', 0)
(2, 'The Best')
```

As with lists and tuples, elements can also be lists and tuples

In [114]:

```
computerLanguages = {
    "Low-level" : ("Assembly", "Machine Code"), #Tuple
    "Procedural" : ("C", "FORTRAAN", "Pascal", "Visual Basic"), #List
    "Object-oriented": ["Java", "C++", "Python"] #List
}
for i in computerLanguages.items():
    print(i)
```

```
('Low-level', ('Assembly', 'Machine Code'))
('Procedural', ('C', 'FORTRAAN', 'Pascal', 'Visual Basic'))
('Object-oriented', ['Java', 'C++', 'Python'])
```

Length of a dictionary

As with lists and tuples, we can use the len( ) function. Note that the length of dictionary is the number of index-element pairs in the dictionary.

In [115]:

```
mixedUp = {
    3.14 : "Pie",
    "e" : 2.7,
    "Stars" : 0,
    2 : "The Best"
}
len(mixedUp)
```

Out[115]:

```
4
```

5. Plotting of graphs

5.1. Basics

We may use the 'pylab' module or the 'matplotlib' module. The functionality of the 'plot' function is similar in both modules.

In [116]:

```
from pylab import plot, show
```

In [117]:

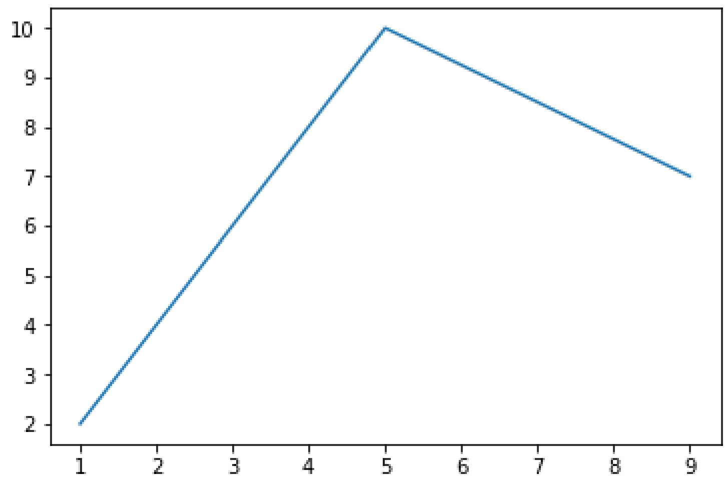
```
x_coords = [1, 5, 9]
y_coords = [2, 10, 7]
```

In [118]:

```
plot(x_coords, y_coords)
```

Out[118]:

```
[<matplotlib.lines.Line2D at 0x10ebf57f0>]
```



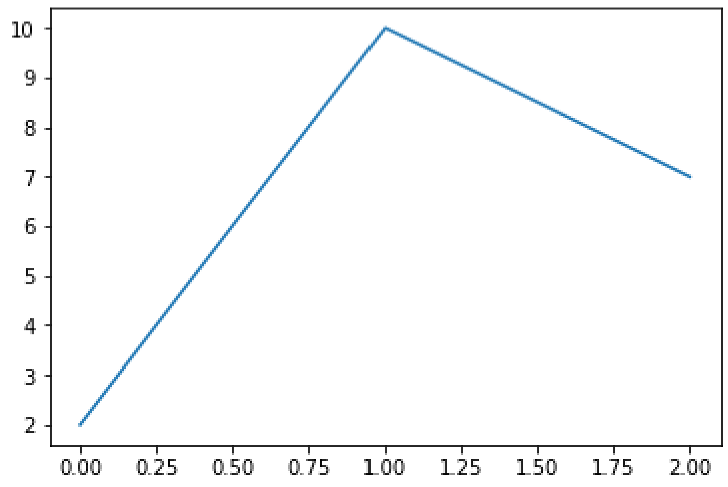
NOTE: If the x values are not specified, then a default range is taken, starting from 0, as seen below

In [119]:

```
plot(y_coords)
```

Out[119]:

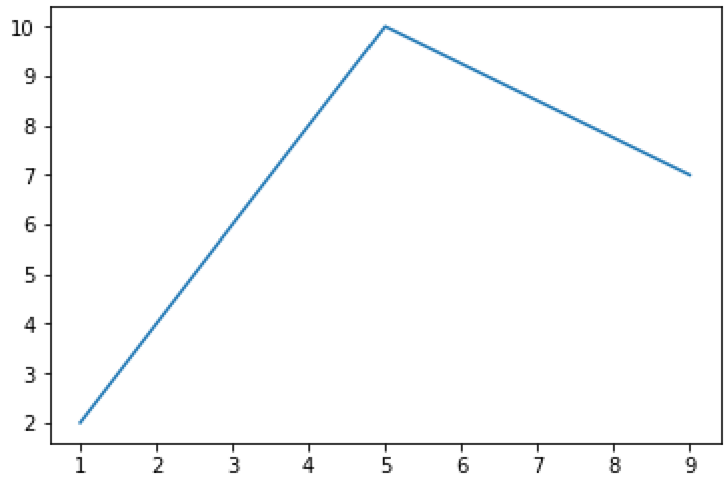
```
[<matplotlib.lines.Line2D at 0x111fb47b8>]
```



The 'show' function presents the graph alone

In [120]:

```
plot(x_coords, y_coords)
show()
```

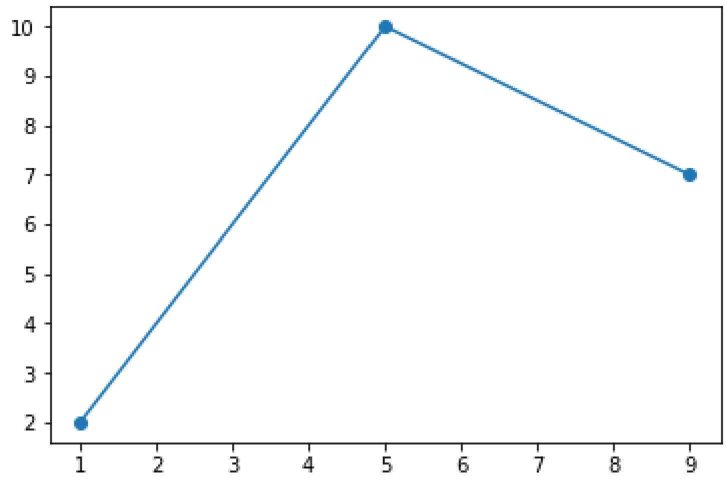


Marking points

To mark the various points plotted by the graph, we can use the 'marker' option in the plot function. For the marker option, use one of the predefined characters, as follows.

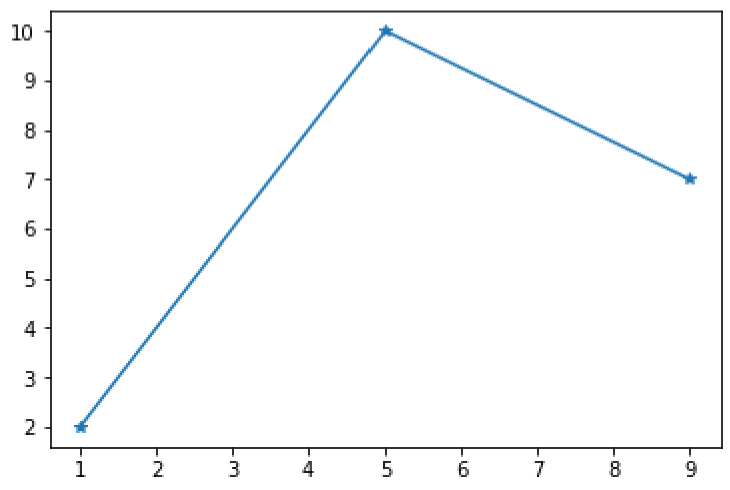
In [121]:

```
plot(x_coords, y_coords, marker = "o")
show()
```



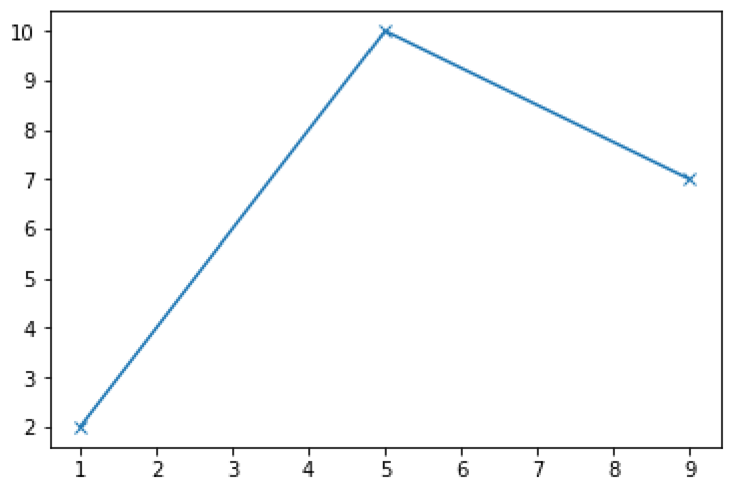
In [122]:

```
plot(x_coords, y_coords, marker = "*")
show()
```



In [123]:

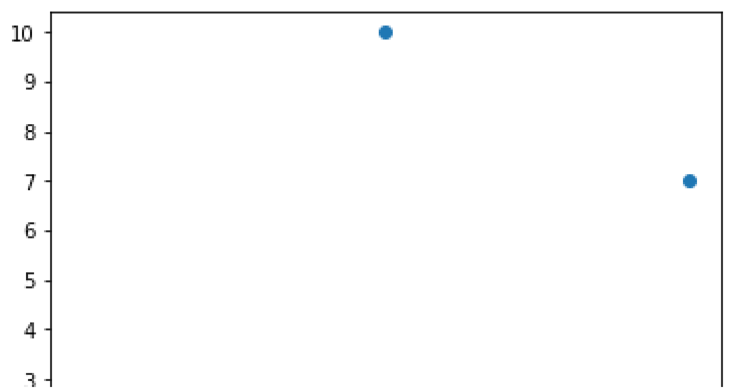
```
plot(x_coords, y_coords, marker = "x")
show()
```

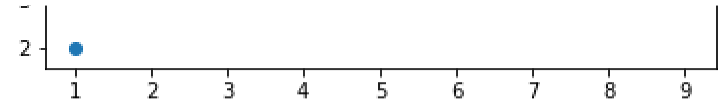


Giving the marker option without typing 'marker = ' results in the plotting of the points alone, as seen below

In [124]:

```
plot(x_coords, y_coords, "o")
show()
```





Graphing temperatures in New York for one year

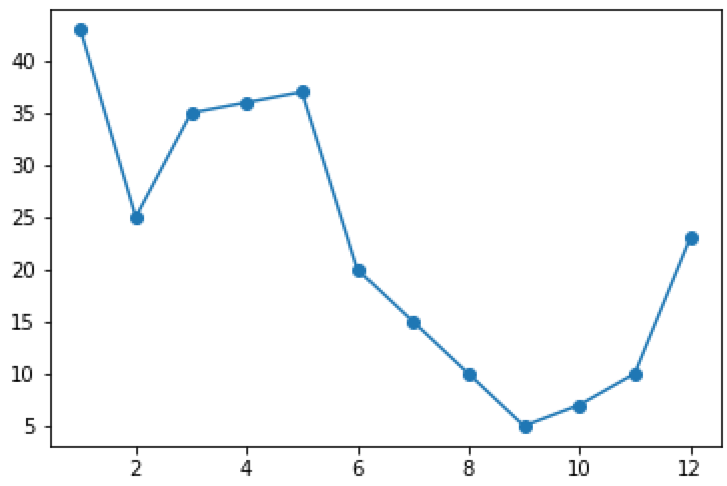
In [125]:

```
months = range(1, 13)
temperatures = [43, 25, 35, 36, 37, 20, 15, 10, 5, 7, 10, 23]
```

In [126]:

```
plot(months, temperatures, marker = 'o')
show(print("Temperatures in NYC"))
```

```
Temperatures in NYC
```



5.2. Multiple plots

In [127]:

```
from pylab import plot, show
```

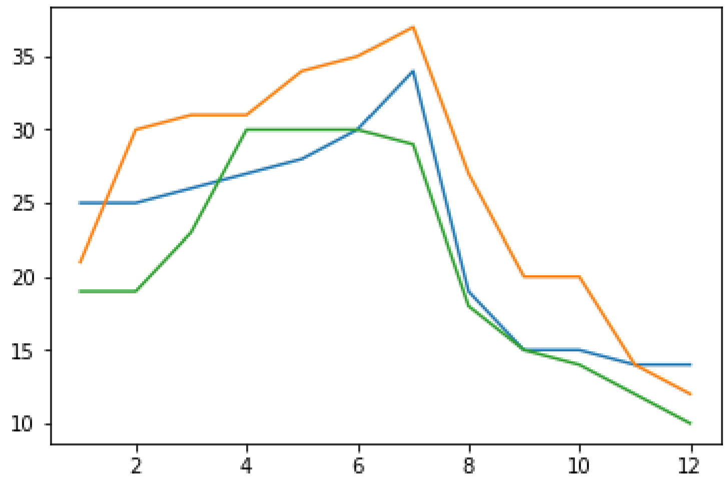
In [128]:

```
year_2001 = [25, 25, 26, 27, 28, 30, 34, 19, 15, 15, 14, 14]
year_2002 = [21, 30, 31, 31, 34, 35, 37, 27, 20, 20, 14, 12]
year_2003 = [19, 19, 23, 30, 30, 30, 29, 18, 15, 14, 12, 10]
```

Method 1

In [129]:

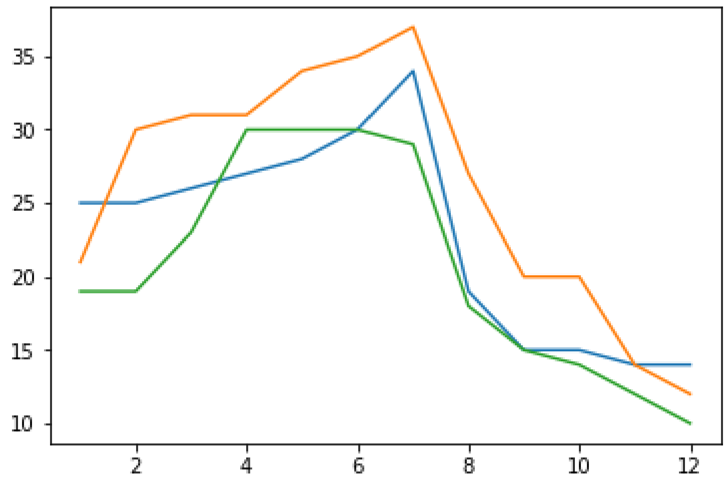
```
plot(range(1, 13), year_2001, range(1, 13), year_2002, range(1, 13), year_200
3)
show()
```



Method 2

In [130]:

```
plot(range(1, 13), year_2001)
plot(range(1, 13), year_2002)
plot(range(1, 13), year_2003)
show()
```



5.3. Customising graphs

Legends

In [131]:

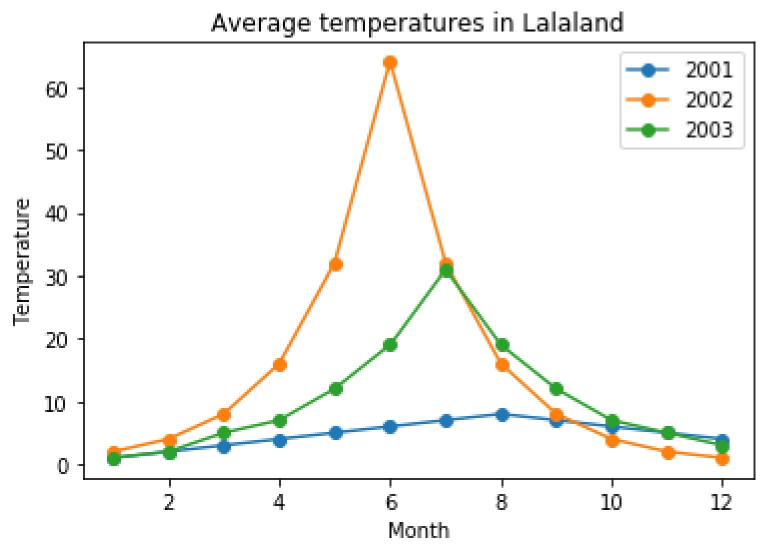
```
from pylab import plot, show, legend, title, xlabel, ylabel
```

In [132]:

```
data1 = [1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4]
data2 = [2, 4, 8, 16, 32, 64, 32, 16, 8, 4, 2, 1]
data3 = [1, 2, 5, 7, 12, 19, 31, 19, 12, 7, 5, 3]
```

In [133]:

```
plot(range(1, 13), data1, range(1, 13), data2, range(1, 13), data3, marker =
'o')
legend([2001, 2002, 2003])
title("Average temperatures in Lalaland")
xlabel("Month")
ylabel("Temperature")
show()
```



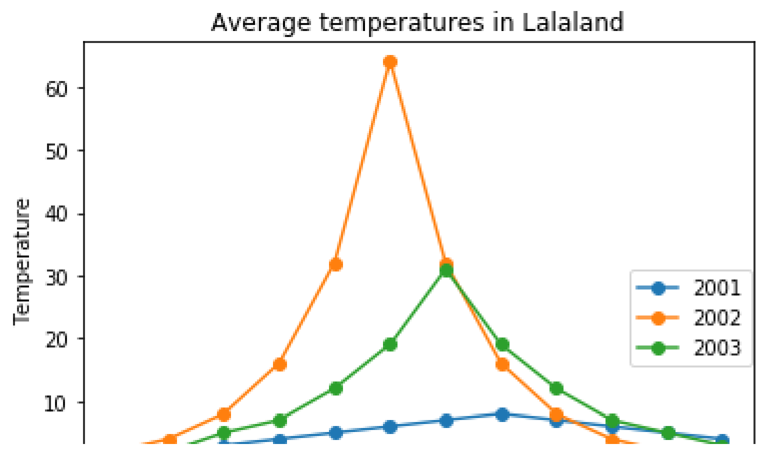
Positioning legends

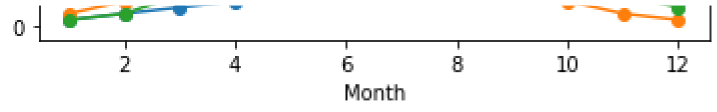
In [134]:

```
from pylab import plot, show, legend, title, xlabel, ylabel
```

In [135]:

```
plot(range(1, 13), data1, range(1, 13), data2, range(1, 13), data3, marker =
'o')
legend([2001, 2002, 2003], bbox_to_anchor = (0.8, 0.5))
title("Average temperatures in Lalaland")
xlabel("Month")
ylabel("Temperature")
show()
```





Grids

In [143]:

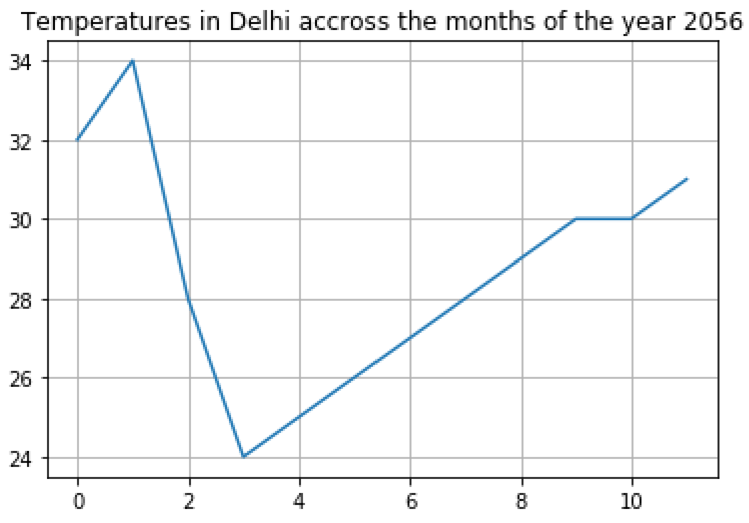
```
from pylab import plot, show, title, grid
```

In [144]:

```
months = range(0, 12)
temperatures = [32, 34, 28, 24, 25, 26, 27, 28, 29, 30, 30, 31]
```

In [145]:

```
plot(temperatures)
title("Temperatures in Delhi accross the months of the year 2056")
grid()
show()
```



Labelling x and y values

To do this, we use xticks and yticks functions...

Labelling x-values

In [146]:

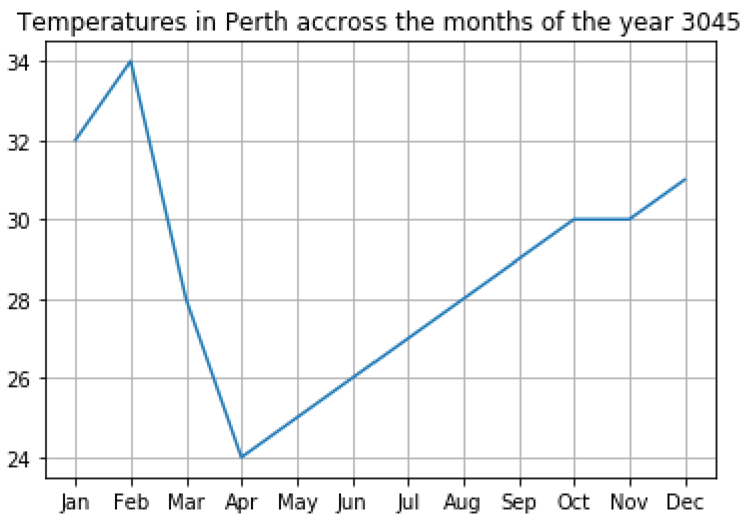
```
from pylab import plot, show, title, xticks, grid
```

In [147]:

```
months = range(0, 12)
monthNames = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
"Oct", "Nov", "Dec"]
temperatures = [32, 34, 28, 24, 25, 26, 27, 28, 29, 30, 30, 31]
```

In [148]:

```
plot(temperatures)
title("Temperatures in Perth accross the months of the year 3045")
xticks(months, monthNames)
grid()
show()
```



Labelling y-values

In [149]:

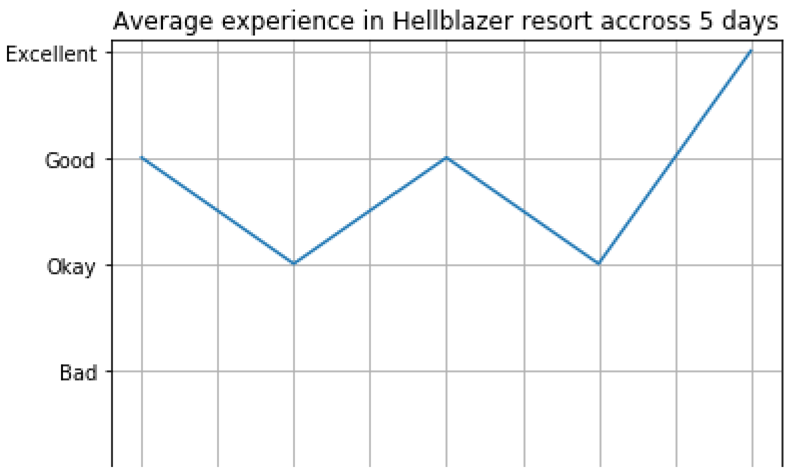
```
from pylab import plot, show, title, yticks, grid
```

In [150]:

```
days = range(0, 5)
ratingScale = range(1, 6)
ratingNames = ["Terrible", "Bad", "Okay", "Good", "Excellent"]
ratings = [4, 3, 4, 3, 5]
```

In [151]:

```
plot(days, ratings)
title("Average experience in Hellblazer resort accross 5 days")
yticks(ratingScale, ratingNames)
grid()
show()
```



Axes

With this function, we can set the minimum of the x or y axes.

In [152]:

```
from pylab import plot, show, axis
```

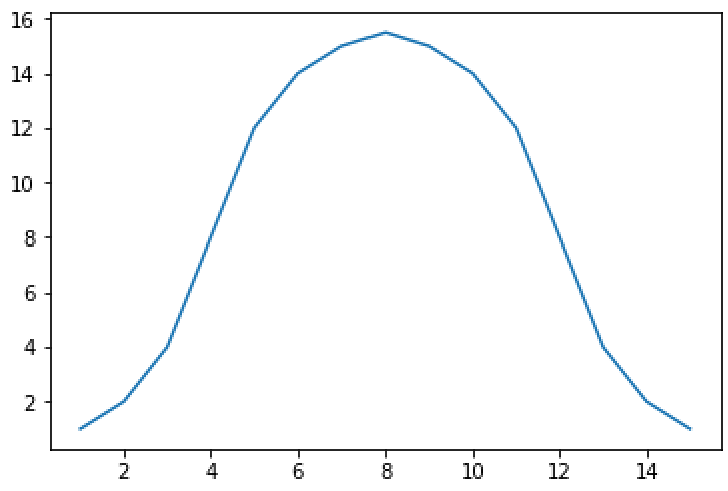
In [153]:

```
data = [1, 2, 4, 8, 12, 14, 15, 15.5, 15, 14, 12, 8, 4, 2, 1]
```

Without axes function...

In [154]:

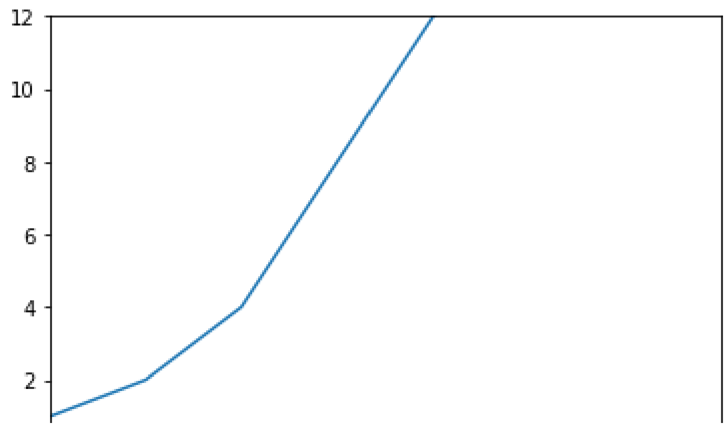
```
plot(range(1, 16), data)
show()
```

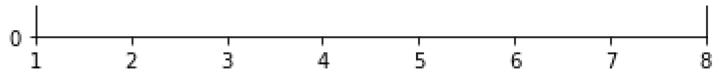


With axes function...

In [155]:

```
plot(range(1, 16), data)
axis([1, 8, 0, 12])
show()
```





It can act as a method to zoom in or zoom out of particular areas of a graph

5.4. Plotting using matplotlib module instead of pylab module

The functionality of the pyplot module in the matplotlib module is almost the same as the pylab functions. The difference is that with matplotlib's pyplot module, it is easier to import with all functionalities (customisation functions) without either individually loading specific functions (as we did with pylab) or unnecessarily loading the memory (which would happen if we loaded the entire pylab module). In usage, the difference is that we need to write the functions in the format moduleName.functionName(...)

In [156]:

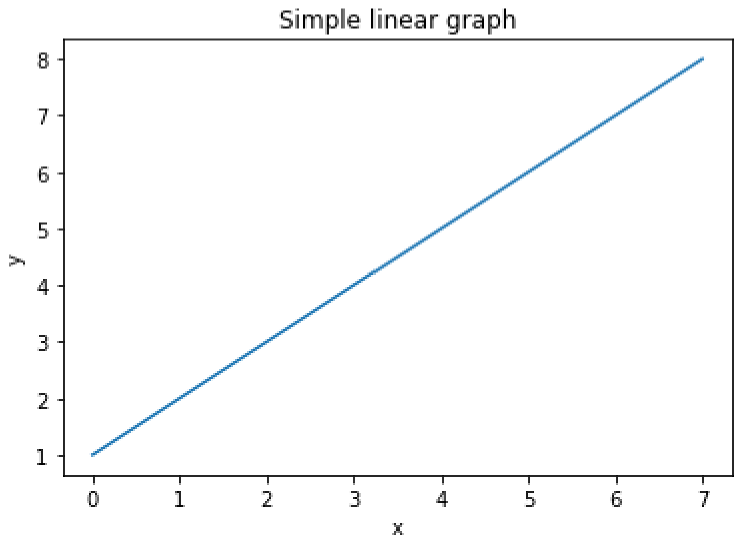
```
import matplotlib.pyplot as plt
```

In [157]:

```
data = [1, 2, 3, 4, 5, 6, 7, 8]
```

In [158]:

```
plt.plot(range(0, 8), data)
plt.title("Simple linear graph")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



5.5. Plotting Newton's formula of gravitation

According to Newton's law of gravitation, the gravitational force exerted on two objects with masses m1 and m2, being apart by a distance of r, is given by $F = Gm_1m_2/r^2$, where G is the gravitational constant.

In [159]:

```
import matplotlib.pyplot as plt
```

Let the masses of the two objects be 1.5 kg and 2.3 kg, with the gravitation constant G being $6.674 * 10^{-11}$

In [160]:

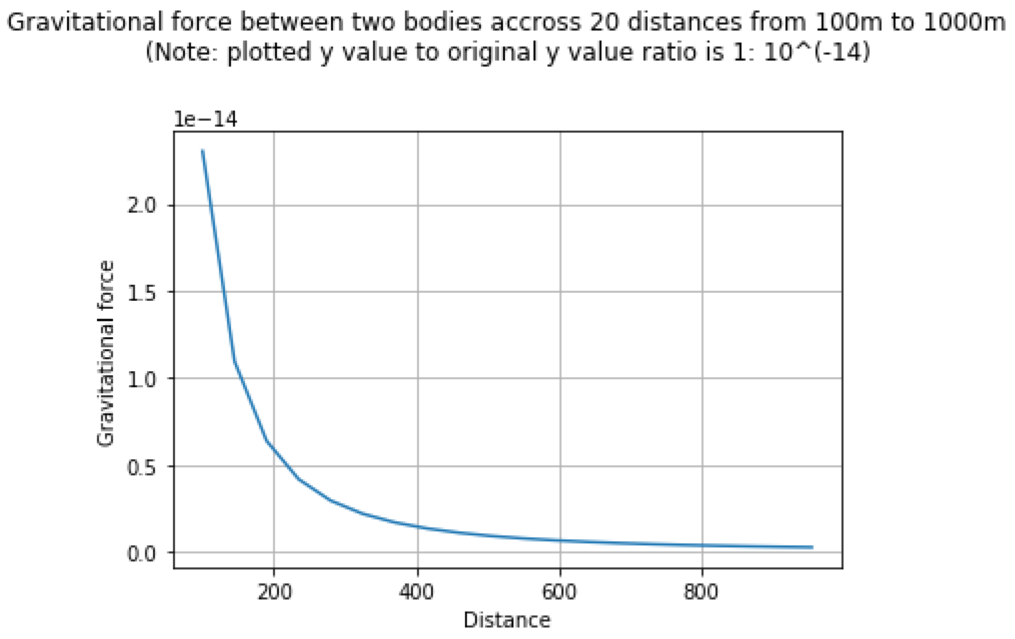
```
m1 = 1.5
m2 = 2.3
G = 6.674 / 10**(11)
```

In [161]:

```
distances = range(100, 1000, int((1000 - 100) /20))
gravitationalForces = []
for i in distances:
    gravitationalForces.append(G * m1 * m2 / i**2)
```

In [162]:

```
plt.plot(distances, gravitationalForces)
plt.title("Gravitational force between two bodies accross 20 distances from 1
00m to 1000m\n(Note: plotted y value to original y value ratio is 1: 10^(-14)
\n\n")
plt.xlabel("Distance")
plt.ylabel("Gravitational force")
plt.grid()
plt.show()
```



5.6. Saving plots

In [163]:

```
from pylab import savefig, plot, show, title, xlabel, ylabel
```

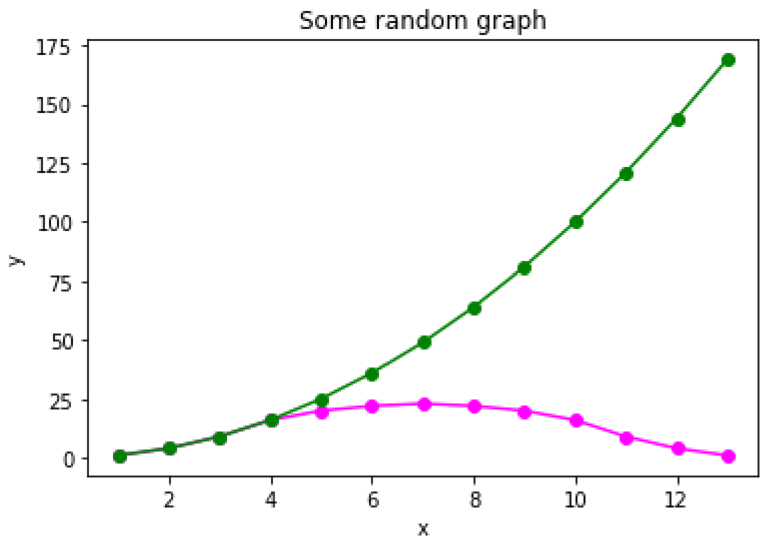
Making some graph...

In [164]:

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
y = [1, 4, 9, 16, 20, 22, 23, 22, 20, 16, 9, 4, 1]
p = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
q = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169]
```

In [165]:

```
plot(x, y, color = 'magenta', marker = 'o')
plot(p, q, color = 'green', marker = 'o')
title("Some random graph")
xlabel("x")
ylabel("y")
savefig("someRandomGraph.pdf")
show()
```



5.7. Projectile motion

Initial velocity = u
Angle between projectile motion and ground = θ
The projectile has two velocity components, horizontal (x) and vertical (y)
x-direction component = $u_x = ucos\theta$
y-direction component = $u_y = sin\theta$

As the projectile moves through space, the velocity changes. We represent the changed velocity as v
x-direction component of velocity change = v_x
y-direction component of velocity change = v_y

Assume v_x does not change, but v_y does due to gravity
$v_y = u_y - gt$, where g is the gravitational constant and and t is the time from launch at which the instantaneous velocity v is measured
We have $u_y = usin\theta$, hence $v_y = usin\theta - gt$

The x-direction component of the projectile motion does not change, so the horizontal distance traveled (S_x) is given by $S_x = v_x t$ => $S_x = u(cos\theta)t$
The y-direction component of the projectile motion does changes, so the vertical distance traveled is

given by the formula $S_y = v_y t$ => $S_y = usin(\theta)t - \frac{1}{2}gt^2$
S_x and S_y are used to calculate the position of the projectile at any given point in time.

Before we write our program, however, we'll need to find out how long the ball will be in flight before it hits the ground so that we know when our program should stop plotting the trajectory of the ball. The ball reaches its highest point when the vertical component of the velocity (v_y) is 0, which is when $v_y = usin\theta - gt = 0$. So we're looking for the value t using the formula $t = \frac{usin\theta}{g}$. We'll call this time t_{peak}.

After it reaches its highest point, the ball will hit the ground after being airborne for another t_{peak} seconds, so the total time of flight (t_{flight}) of the ball is $t_{flight} = 2\frac{usin\theta}{g}$

Let's take a ball that's thrown with an initial velocity (u) of $5m/s$ at an angle (θ) of 45 degrees. To calculate the total time of fight, we substitute $u = 5; \theta = 45$, and $g = 9.8$ into the equation we saw above. The ball will be in air for this period of time, so to draw the trajectory, we'll calculate its x- and y-coordinates at regular intervals during this time period.

In [166]:

```
from matplotlib import pyplot as plt
import math
```

The range( ) function can only create ranges with integer increments, so we need to make our own function that makes a range with fractional increments.
(We need fractional increments because drawing the graph with consequent x-values sufficiently close together will make a smoother graph)

In [167]:

```
def frange(start, final, increment):
    numbers = []
    while start < final:
        numbers.append(start)
        start = start + increment
    return numbers
```

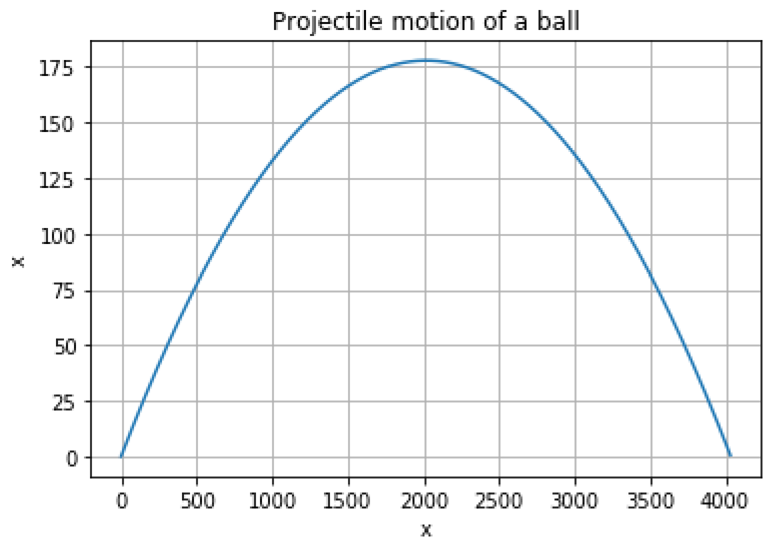
In [168]:

```
def drawTrajectory(u, theta):
    theta = math.radians(theta) # Converting initial angle of the trajectory
 to radians
    g = 9.8 # Gravitational constant (acceleration due to gravity on Earth)
    t_flight = 2 * u * math.sin(theta) / g # Total time the ball is in the ai
r
    intervals = frange(0, t_flight, 0.01) # To make a smoother graph
    x = []
    y = []
    for t in intervals:
        x.append(u * math.cos(theta) * t)
        y.append(u * math.sin(theta) * t - 0.5 * g * t * t)
    plt.plot(x, y)
    plt.xlabel("x")
    plt.ylabel("x")
    plt.grid()
    plt.title("Projectile motion of a ball")
```

In [169]:

```
try:
    u = float(input("Enter the initial velocity (m/s): "))
    theta = float(input("Enter the angle of projection (degrees):"))
except:
    print("You entered an invalid input")
else:
    drawTrajectory(u, theta)
    plt.show()
```

```
Enter the initial velocity (m/s): 340
Enter the angle of projection (degrees):10
```



5.8. Bar plot

In [170]:

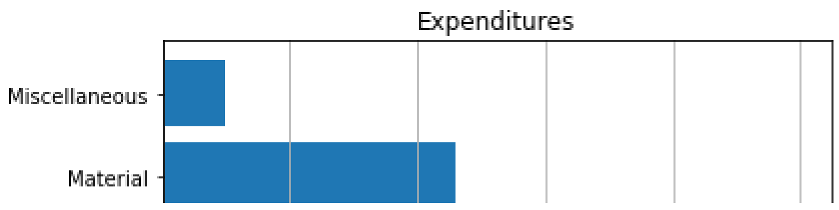
```
import matplotlib.pyplot as plt
```

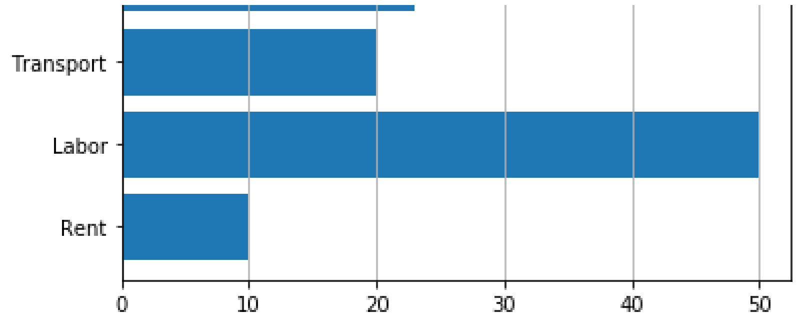
In [171]:

```
data = [10, 50, 20, 23, 5]
positions = range(1, len(data) + 1)
labels = ["Rent", "Labor", "Transport", "Material", "Miscellaneous"]
```

In [172]:

```
plt.barh(positions, data, align = 'center')
plt.title("Expenditures")
plt.yticks(positions, labels)
plt.grid(axis = 'x', linestyle = '-')
# axis = x means vertical gridlines will be drawn from each x-value
# axis = y means horizontal gridlines will be drawn from each y-value
plt.show()
```





6. Working with symbols

Working with symbols is essential for algebraic computation. To work with symbols, we use the module sympy.

6.1. Variable symbols

In [173]:

```
from sympy import Symbol
```

In [174]:

```
x = Symbol('x')
myExpression = x**2 + 2*x + 3
print(myExpression)
```

```
x**2 + 2*x + 3
```

6.2. Pretty printing

Pretty printing prints an expression in a manner closer to the way we write Mathematics in practice. The function for pretty printing is pprint( ).

In [175]:

```
from sympy import Symbol, pprint
```

In [176]:

```
y = Symbol('y')
myExpression = 5*y**3 + 2*y**2 + 3*y + 9
pprint(myExpression)
```

```
   3      2
5⋅y  + 2⋅y  + 3⋅y + 9
```

6.3. Operations on an expression

Adding and subtracting terms

In [177]:

```
from sympy import Symbol, pprint
```

In [178]:

```
x = Symbol('x')
myExpression = 5*x**3 + 2*x**2
print("Old expression:")
pprint(myExpression)

newExpression = myExpression + 3*x - 9
print("New expression:")
pprint(newExpression)
```

```
Old expression:
   3      2
5⋅x  + 2⋅x 
New expression:
   3      2
5⋅x  + 2⋅x  + 3⋅x - 9
```

You can also actually add and subtract terms, rather than merely append them to the expression...

In [179]:

```
x = Symbol('x')
myExpression = 5*x**3 + 2*x**2
print("Old expression:")
pprint(myExpression)

newExpression = myExpression + 3*x**3 - 2*x**2
print("New expression:")
pprint(newExpression)
```

```
Old expression:
   3      2
5⋅x  + 2⋅x 
New expression:
   3
8⋅x 
```

Multiplying and dividing terms

In [180]:

```
from sympy import Symbol, pprint
```

In [181]:

```
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
myExpression = 5*x**3 + 2*x**2
print("Old expression:")
pprint(myExpression)

newExpression = myExpression * (3*y**3 - 2*y**2) / (3*z - 4)
```

```
print("New expression:")
pprint(newExpression)
```

```
Old expression:
   3      2
5⋅x  + 2⋅x
New expression:
⎛   3      2⎞ ⎛   3      2⎞
⎝5⋅x  + 2⋅x ⎠⋅⎝3⋅y  - 2⋅y ⎠
───────────────────────────
          3⋅z - 4
```

You can also multiply and divide terms with constants...

In [182]:

```
x = Symbol('x')
myExpression = 4*x + 2
print("Expression:")
pprint(myExpression)

newExpression = myExpression * 8
print("Expression X 8:")
pprint(newExpression)

newExpression = myExpression / 4
print("Expression / 4:")
pprint(newExpression)
```

```
Expression:
4⋅x + 2
Expression X 8:
32⋅x + 16
Expression / 4:
x + 1/2
```

Multiplying and dividing powers of the same symbol changes the powers of the symbol accordingly...

In [183]:

```
x = Symbol('x')
myExpression = x**3
print("Expression: ")
pprint(myExpression)

newExpression = myExpression * x**2
print("Expression * x^2: ")
pprint(newExpression)

newExpression = myExpression / x**2
print("Expression / x^2: ")
pprint(newExpression)
```

```
Expression:
 3
x
Expression * x^2:
 5
x
Expression / x^2:
x
```

6.4. Substituting values into symbols in an expression

To subsitute values into symbols in an expression, we use the subs( ) function
This function has two arguments: symbol, value
The function must be called by the expression object

In [184]:

```
from sympy import Symbol, pprint
```

In [186]:

```
x = Symbol('x')
myExpression = x**3 + 5*x**(0.5) - x**(-3)
print("My expression: ")
pprint(myExpression)

n = float(input("Enter the value of x for which you want to evaluate the expr
ession: "))
myExpression.subs(x, n)
```

```
My expression:
   0.5    3   1
5⋅x    + x  - ──
               3
              x
Enter the value of x for which you want to evaluate the expressio
n: 4.86
```

Out[186]:

```
125.805248377602
```

6.5. Making a series

Write the series $y = \sum_{n=1}^{\infty} = \frac{(-1)^n x^{2n+1}}{2n+1}$ for 10 terms. We can see that the terms of the series are $-\frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \ldots$

In [187]:

```
from sympy import Symbol, pprint
```

In [188]:

```
x = Symbol('x')
mySeries = -(x**3)/3
for i in range (2, 11):
    mySeries = mySeries + ((-1)**i) * (x**(2*i + 1)) / (2*i + 1)
pprint(mySeries)
```

```
 21     19     17     15     13     11     9     7     5     3
x      x      x      x      x      x      x     x     x     x
─── - ─── + ─── - ─── + ─── - ─── + ── - ── + ── - ──
 21     19     17     15     13     11    9     7     5     3
```

6.6. Re-ordering a series of terms using init_printing( )

Normally, the terms of a series of terms are ordered with the highest power first. But while this is the standard for polynomials, series need to be arranged with the lowest powers first. To reverse the default order, we use the init_printing( ) function, and apply the option: order = "rev-lex"

In [189]:

```
from sympy import Symbol, pprint, init_printing
```

Let us reverse the order of the series in the previous section...

In [190]:

```
init_printing(order = 'rev-lex')
x = Symbol('x')
mySeries = -(x**3)/3
for i in range (2, 11):
    mySeries = mySeries + ((-1)**i) * (x**(2*i + 1)) / (2*i + 1)
pprint(mySeries)
```

```
   3    5    7    9     11     13     15     17     19     21
  x    x    x    x     x      x      x      x      x      x
- ── + ── - ── + ── - ─── + ─── - ─── + ─── - ─── + ───
  3    5    7    9     11     13     15     17     19     21
```

6.7. Plotting a series

Here, we write a Python program to calculate sine function using series expansion and plot it
Maclauring expansion of sin(x) is
$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$

In [191]:

```
from sympy import Symbol
import matplotlib.pyplot as plt
```

The factorial finding function...

In [192]:

```
def factorial(n):
    fact = int(1)
    for i in range(2, n + 1):
        fact = fact * i
    return fact
```

Making the series for 100 terms

In [193]:

```
x = Symbol('x')
sine = x
for i in range(1, 100):
```

```
# I chose 101 because the accuracy does not improve anymore
# for 16 decimal places after i = 100,
# for a sufficient range of x values
    sine = sine + ((-1)**i) * (x**(2*i + 1)) / factorial(int(2*i + 1))
```

Defining the x and y values...
x-values are defined as the interval [0, 2], with the distance between consequent x-values being 0.1
y-values are defined by substituting each x-value into the obtained series

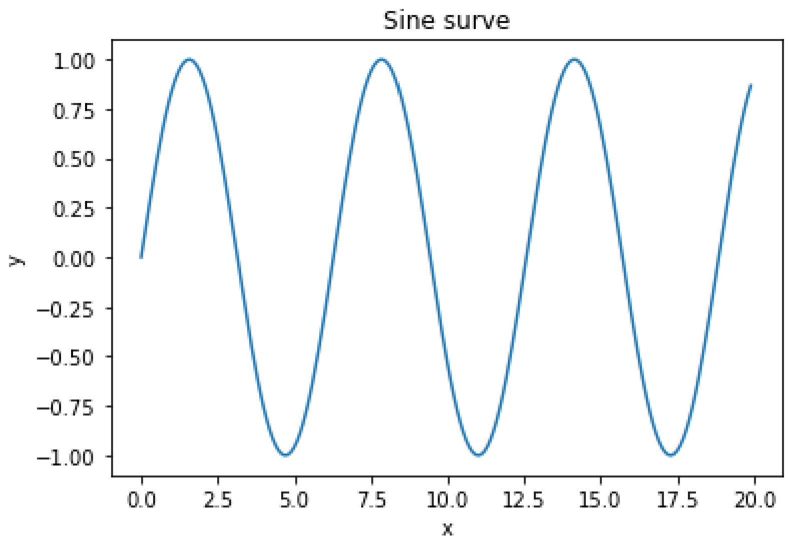
In [194]:

```
x_values = []
y_values = []
for i in range(0, 200, 1):
    n = float(i) / 10
    x_values.append(n)
    y_values.append(sine.subs(x, n))
```

Plotting the curve...

In [195]:

```
plt.plot(x_values, y_values)
plt.title("Sine surve")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



7. The format method

This method is applicable for strings, and is used to insert separate arguments into the string. The return value of this method is string.

7.1. Some examples of usage

In [196]:

```
item1 = "orange"
```

```
item2 = "apple"
item3 = "mango"
print("At the grocery store, I bought some {0}, {1} and {2}.".format(item1, i
tem2, item3))
```

```
At the grocery store, I bought some orange, apple and mango.
```

In [197]:

```
print("Here we go in {0}, {1}, {2}, {3}!".format(3, 2, 1, "go"))
```

```
Here we go in 3, 2, 1, go!
```

In [198]:

```
print("This is version {0}, called {1}, and its model number is {2}.".format(
2.0, "DoomSlayer", 10))
```

```
This is version 2.0, called DoomSlayer, and its model number is 1
0.
```

7.2. Applied on variable strings

Demonstration that the method is applicable for any string...

In [199]:

```
myString = "My name is {0} {1}, and my roll number is {2} and my class is {3}
."
print(myString.format("Pranav", "Gopalkrishna", 1940223, "3 CMS"))
```

```
My name is Pranav Gopalkrishna, and my roll number is 1940223 and
my class is 3 CMS.
```

7.3. Applied without print( )

Without the print function, calling the format method simply returns a string

In [200]:

```
"Here we go in {0}, {1}, {2}, {3}.".format(1, 2, 3, "go")
```

Out[200]:

```
'Here we go in 1, 2, 3, go.'
```

In [201]:

```
myString.format("Pranav", "Gopalkrishna", 1940223, "3 CMS")
```

Out[201]:

```
'My name is Pranav Gopalkrishna, and my roll number is 1940223 an
d my class is 3 CMS.'
```

7.4. Applied in different orders

In [202]:

```
"Here we go in {3}, {2}, {1}, {0}.".format("go", 1, 2, 3)
```

Out[202]:

```
'Here we go in 3, 2, 1, go.'
```

In [203]:

```
"{2}, {2}, {1}, {1}, {0}, {0}".format("HO", "HE", "HU")
```

Out[203]:

```
'HU, HU, HE, HE, HO, HO'
```

7.5. Repeating values

In [204]:

```
"{0}".format("HA"*4)
```

Out[204]:

```
'HAHAHAHA'
```

In [205]:

```
"{0}".format("HA" + "HA")
```

Out[205]:

```
'HAHA'
```

The repetition won't work for numbers

In [206]:

```
"{0}".format(3 * 2)
```

Out[206]:

```
'6'
```

In [207]:

```
"{0}".format(3.2 + 4.5)
```

Out[207]:

```
'7.7'
```

7.6. Rounding decimals

In the brackets { }, write {order_no:.num_of_decimal_places f)
Example: {1:.3f}

In [208]:

```
"{0:.2f}".format(3.237)
```

Out[208]:

```
'3.24'
```

In [209]:

```
"{0:.5f}".format(2)
```

Out[209]:

```
'2.00000'
```

8. Roots of a quadratic equation

Let general quadratic equation be $ax^2 + bx + c = 0$. Let the roots of this equation be
$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$
$x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$

Comparing the general equation to $2x^2 + 4x + 7 = 0$, we have
$a = 2, b = 4, c = 7$

To solve the equation, we store the values of a, b and c. Then we substitute these values in the determinants, i.e. the values within the square root of x_1 and x_2, i.e. $b^2 - 4ac$. If determinant is positive, then proceed to find x_1 and x_2. If determinant is negative, then find $\frac{-b}{2a}$ separately as r, and write the roots as complex numbers $r + di$ and $r - di$, where d is the determinant and i is $\sqrt{-1}$.

Quadratic equation root finder function defintion...

In [210]:

```
from math import sqrt
```

In [211]:

```
def roots(a, b, c):
    a = float(a)
    b = float(b)
    c = float(c)
    d = float(b*b - 4*a*c)
    if d < 0:
        print("Root 1: {0} + {1}i".format(-b/2*a, sqrt(-d)/2*a))
        print("Root 2: {0} - {1}i".format(-b/2*a, sqrt(-d)/2*a))
    else:
        print("Root 1: {0}".format((-b + sqrt(d))/2*a))
        print("Root 2: {0}".format((-b - sqrt(d))/2*a))
```

Finding the roots of a quadratic equation, given the three coefficients...

In [213]:

```
a = float(input("Enter coefficient of x^2: "))
b = float(input("Enter coefficient of x: "))
```

```
c = float(input("Enter constant: "))
roots(a, b, c)
```

```
Enter coefficient of x^2: 45
Enter coefficient of x: 23
Enter constant: 67
Root 1: -517.5 + 2416.1061131498345i
Root 2: -517.5 - 2416.1061131498345i
```

9. Converting units of measurement

The international system of units defines 7 base quantities. Length, mass, time and temparature are 4 of the 7 based quantities, with their standard units of measurements being meters, grams, seconds and kelvin respectively.

1 inch = 2.54 centimeters

1 mile = 1.609 km

Celsius = (Fahrenheit - 32)*5/9

Fahrenheit = Celsius*9/5 + 32

9.1. Program to convert miles to kilometers or vice versa

In [214]:

```python
def options():
    print("Enter 1 to convert kilometers to miles")
    print("Enter anything else to convert miles to kilometers")
    return input()
```

In [215]:

```python
def km_to_mil():
    km = float(input("Enter no. of kilometers: "))
    print("{0} km = {1:.3f} mil".format(km, km * 1.609))
def mil_to_km():
    mil= float(input("Enter no. of miles: "))
    print("{0} mil = {1:.3f} km".format(mil, mil / 1.609))
```

In [216]:

```python
def km_mil_converter():
    option = options()
    if option == "1":
        km_to_mil()
    else:
        mil_to_km()
```

In [217]:

```python
km_mil_converter()
```

```
Enter 1 to convert kilometers to miles
Enter anything else to convert miles to kilometers
Enter anything else to convert miles to kilometers
1
Enter no. of kilometers: 100
100.0 km = 160.900 mil
```

9.2. Enhanced unit converter

In [218]:

```python
def moreOptions():
    print("Enter 1 to convert kilometers to miles")
    print("Enter 2 to convert miles to kilometers")
    print("Enter 3 to convert Celsius to Fahrenheit")
    print("Enter 4 to convert Fahrenheit to Celsius")
    print("Enter 5 to convert kilograms to pounds")
    print("Enter 6 to convert pounds to kilograms")
    return input()
```

In [219]:

```python
def km_to_mil():
    km = float(input("Enter no. of kilometers: "))
    print("{0} km = {1:.3f} mil".format(km, km * 1.609))
def mil_to_km():
    mil= float(input("Enter no. of miles: "))
    print("{0} mil = {1:.3f} km".format(mil, mil / 1.609))
```

In [220]:

```python
def celsius_to_fahr():
    c = float(input("Enter degrees Celsius: "))
    print("{0}°C = {1:.3f}°F".format(c, c * 9 / 5 + 32))
def fahr_to_celsius():
    f = float(input("Enter degrees Fahrenheit: "))
    print("{0}°C = {1:.3f}°F".format(f, (f - 32) * 5 / 9))
```

In [221]:

```python
def kilo_to_pound():
    k = float(input("Enter weight in kilograms: "))
    print("{0} kg = {1:.3f} pound".format(k, k * 2.205))
def pound_to_kilo():
    p = float(input("Enter weight in pounds: "))
    print("{0} pound = {1:.3f} kg".format(p, p / 2.205))
```

In [222]:

```python
def enhancedConverter():
    option = moreOptions()
    if option == "1":
        km_to_mil()
    elif option == "2":
        mil_to_km()
    elif option == "3":
        celsius_to_fahr()
    elif option == "4":
        fahr_to_celsius()
    elif option == "5":
        kilo_to_pound()
    elif option == "6":
        pound_to_kilo()
    else:
        print("Invalid option")
```

In [223]:

```
enhancedConverter()
```

```
Enter 1 to convert kilometers to miles
Enter 2 to convert miles to kilometers
Enter 3 to convert Celsius to Fahrenheit
Enter 4 to convert Fahrenheit to Celsius
Enter 5 to convert kilograms to pounds
Enter 6 to convert pounds to kilograms
3
Enter degrees Celsius: 100
100.0°C = 212.000°F
```

10. Exception handling

An exception is an unwanted event occurring in a program's execution that disrupts the algorithmic execution of the program. To handle exceptions, we use the try and except statements. In the following example, we apply exception handling to the multiplication table to ensure that the input is a number.

In [224]:

```
def enhancedMultiplicationTable():
    try:
        num = float(input("Enter a number to find its multiplication table: "
))
        lim = int(input("Enter the factor until which you want to find multip
les: "))
    except:
        print("Invalid inputs! You must enter a number for the first, and an
 integer for the second")
        return
    for i in range(1, lim + 1):
        print("{0}\tX\t{1}\t=\t{2}".format(num, i, num * i))
```

In [227]:

```
enhancedMultiplicationTable()
```

```
Enter a number to find its multiplication table: abc
Invalid inputs! You must enter a number for the first, and an int
eger for the second
```

In [228]:

```
enhancedMultiplicationTable()
```

```
Enter a number to find its multiplication table: 10.4
Enter the factor until which you want to find multiples: 10.4
Invalid inputs! You must enter a number for the first, and an int
eger for the second
```

In [229]:

```
enhancedMultiplicationTable()
```

```
Enter a number to find its multiplication table: 10.4

Enter the factor until which you want to find multiples: 10
10.4    X       1       =       10.4
10.4    X       2       =       20.8
10.4    X       3       =       31.200000000000003
10.4    X       4       =       41.6
10.4    X       5       =       52.0
10.4    X       6       =       62.400000000000006
10.4    X       7       =       72.8
10.4    X       8       =       83.2
10.4    X       9       =       93.60000000000001
10.4    X       10      =       104.0
```