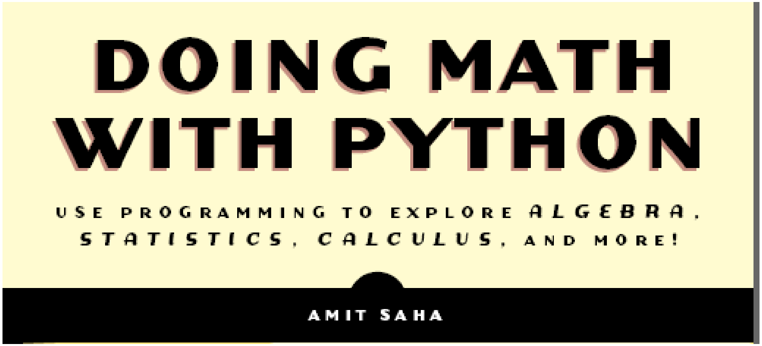
Introduction to Python Programming

August 21, 2018

Working with fractions

Contents are taken from the textbook *Amit Saha, Doing Maths With Python, no starch press, Sanfrancisco*:



DOING MATH WITH PYTHON

USE PROGRAMMING TO EXPLORE ALGEBRA, STATISTICS, CALCULUS, AND MORE!

AMIT SAHA

Working with fractions

- To work with fractions we need to use Pythons fraction module
- Module is a program written by some one else, that we can use in our own programs
- fractions module is a part of the standard library meaning that it is already installed
- Before we need to use it, we'll import it
- Python displays the fraction in the form Fraction (numerator, denominator)

Example

```
from fractions import Fraction
f = Fraction (8,9)
f
Fraction(8,9)
```

Working with fractions

Example

```
Fraction(3,4)+1+1.5
3.25
```

- If the input is a fraction or an integer, then the output is a fraction

Example

```
Fraction(3,4)+1+Fraction(1,4)
Fraction(2,1)
```

Working with Complex Numbers

- Python supports complex numbers with imaginary part identified by the letter j or J.
- The complex number $4 + 5i$ would be written in Python as $4 + 5j$.

Example

```
a = complex(6, 8)
a
6+8j
b = complex (8,3)
b
8+3j
a+b
14+11j
```

Working with Complex Numbers

Example

a - b
-2 + 5j

- Multiplication and division of complex numbers

Example

a*b
24 + 82j
a/b
(0.9863013698630136+0.6301369863013698j)

Working with Complex Numbers

Example

```
a - b
-2 + 5j
```

- The modulus % and the floor division // are not valid for complex numbers
- The real and imaginary part of a complex number can be retrieved

Example

```
z = 2 + 3j
z.real
2.0
z.imag
3.0
```

Working with Complex Numbers

- Conjugate of a complex number has the same real part but an imaginary part with an equal magnitude and an opposite sign

Example

z.conjugate()
2-3j

Magnitude of a complex number
$\sqrt{x^2 + y^2}$ Using the real and imaginary part.

Example

$(z.real **2 + z.imag **2) **0.5$

Magnitude of a complex number

Example

3.605551275463989
Simpler Way: absolute value of a function $abs(z)$
3.6055512754639896

- In python a string is any set of characters between two quotes:
- To create a string, either single quotes or double quotes can be used.

Example

$S_1 = \ 'a\ string'$
$S_2 = \ ''a\ string''$

- a string can be converted into an integer or floating point number using $int()$ or $float()$ function

String

Example

```
a = ' 1 '
int(a)+1
2
float(a)+1
2.0
```

- If we take a string that has a floating point number (like 3.5 or even 4.0) and input that string into *int*() function, then we will get an error message

Handling invalid input

Example

```
int('4.0')
Traceback (most recent call last): ValueError: invalid literal for int()
with base 10: '4.0'
```

Complex number as an input

Example

```
z = complex(input('Enter a complex number:'))
Enter a complex number: 2+3j
out[] 2+3j
```

Is 2.0.integer

- *is_integer*() method to filter out any numbers with a significant digit after the decimal point. (This method is only defined for float type numbers in Python; it wont work with numbers that are already entered in integer form.)

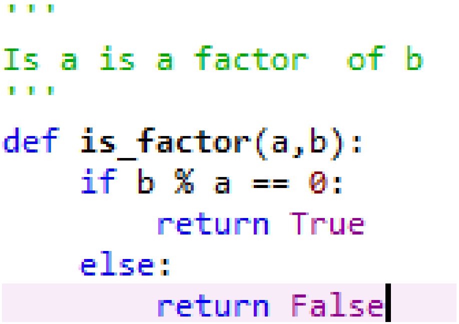
Example

```
4.6.is_integer()
False
4.0.is_integer()
True
```

Program1: Is a is a factor of b

- When a non zero integer a, divides another integer b, leaving a remainder 0, then a is said to be a factor of b.
- **Question:** Is 2 is a factor of 36 ?

Program to check the above question



```
'''
Is a is a factor  of b
'''
def is_factor(a,b):
    if b % a == 0:
        return True
    else:
        return False
```

Program1: Is a is a factor of b

- Three straight single quotes ('). The text in between those quotes wont be executed by Python as part of the program; its just commentary for us humans.

Example

$is\_factor(2, 36)$
True

How range works

- A typical use of the *range*() function looks like this:

Example

```
for i in range (1,6):
    print(i)
1
2
3
4
5
```

How range works

- We can also use the *range*() function without specifying the start value,

Example

```
for i in range (6):
    print(i)
0
1
2
3
4
5
```

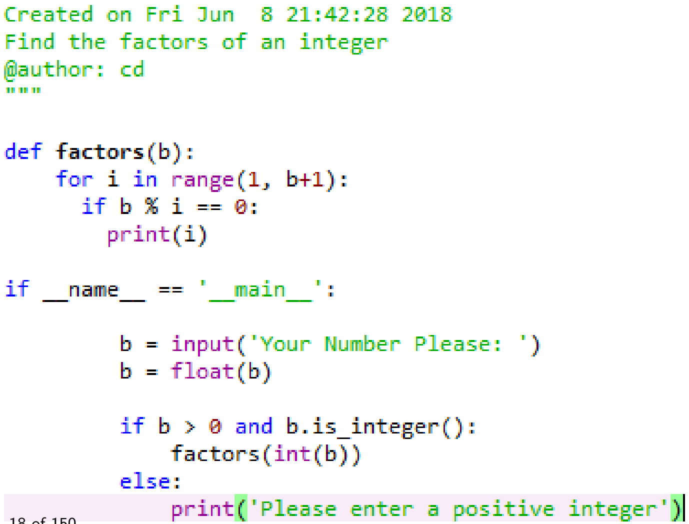
How range works

- The difference between two consecutive integers produced by the *range*() function is known as the step value. By default, the step value is 1.
- To specify a different step value, specify it as the third argument (the start value is not optional when you specify a step value). For example, the following program prints the odd numbers below 10:

Example

```
for i in range (1,10,2):
    print(i)
1
3
5
7
9
```

Calculating the Factors of an Integer



```python
Created on Fri Jun  8 21:42:28 2018
Find the factors of an integer
@author: cd
"""

def factors(b):
    for i in range(1, b+1):
      if b % i == 0:
        print(i)

if __name__ == '__main__':

         b = input('Your Number Please: ')
         b = float(b)

         if b > 0 and b.is_integer():
             factors(int(b))
         else:
             print('Please enter a positive integer')
```

Generating Multiplication Tables

- $if\ \_\_name\_\_ ==' main'$ This block of code ensures that the statements within the block are executed only when the program is run on its own.
- **Generating Multiplication Tables**
- Consider three numbers, a, b, and n, where n is an integer, such that $a \times n = b$. We can say here that b is the n^{th} multiple of a. For example, 4 is the 2nd multiple of 2, and 1024 is the 512nd multiple of 2.
- Our next program is to generate multiplication table.
- In this program, well use the $format()$ method with the $print()$ function to help make the programs output look nicer and more readable.

Generating Multiplication Tables

- *format*() method

Example

item1 = 'orange'
item2 = 'apple'
item3 = 'grapes'
print('At the grocery store, Luis bought some
$\{0\}, \{1\} and \{2\}'$.format(item1, item2, item3))
At the grocery store, Luis bought some orange, apple and grapes.

Generating Multiplication Tables

- First, we created three labels (item1, item2, and item3), each referring to a different string (orange, apple, and grapes). Then, in the print() function, we typed a string with three placeholders in curly brackets: $\{0\}, \{1\}, and \{2\}$.
- We followed this with .format(), which holds the three labels we created. This tells Python to fill those three placeholders with the values stored in those labels in the order listed, so Python prints the text with $\{0\}$ replaced by the first label, $\{1\}$ replaced by the second label, and so on.

Generating Multiplication Tables

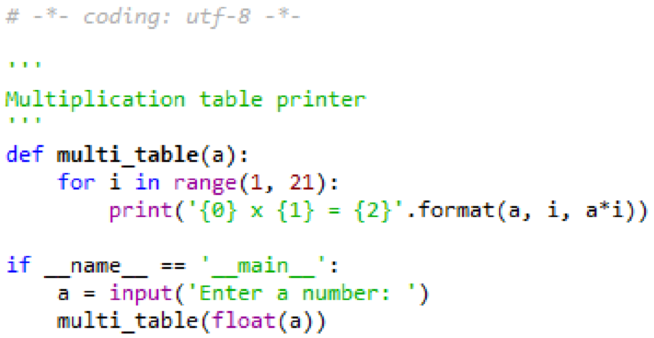
- Its not necessary to have labels pointing to the values we want to print. We can also just type values into .format(), as in the following example:

Example

$print('Number1 : \{0\}Number2 : \{1\}'.format(56.1, 4.578))$

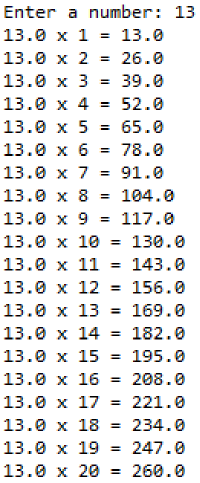
```
Number 1: 567.5 Number 2: 4.578
```

Generating Multiplication Tables



```python
# -*- coding: utf-8 -*-

'''
Multiplication table printer
'''
def multi_table(a):
    for i in range(1, 21):
        print('{0} x {1} = {2}'.format(a, i, a*i))

if __name__ == '__main__':
    a = input('Enter a number: ')
    multi_table(float(a))
```

Generating Multiplication Tables



OUTPUT:

```
Enter a number: 13
13.0 x 1 = 13.0
13.0 x 2 = 26.0
13.0 x 3 = 39.0
13.0 x 4 = 52.0
13.0 x 5 = 65.0
13.0 x 6 = 78.0
13.0 x 7 = 91.0
13.0 x 8 = 104.0
13.0 x 9 = 117.0
13.0 x 10 = 130.0
13.0 x 11 = 143.0
13.0 x 12 = 156.0
13.0 x 13 = 169.0
13.0 x 14 = 182.0
13.0 x 15 = 195.0
13.0 x 16 = 208.0
13.0 x 17 = 221.0
13.0 x 18 = 234.0
13.0 x 19 = 247.0
13.0 x 20 = 260.0
```

Generating Multiplication Tables

- We can use the format() method to further control how numbers are printed. For example, if we want numbers with only two decimal places, we can specify that with the format() method. Here is an example:

Example

$'\{0\}'.format(1.25356)$
$'1.25356'$
$'\{0:.2f\}'.format(1.25356)$
$'1.25'$

Generating Multiplication Tables

- $\{0 : .2f\}$, meaning that we want only two numbers after the decimal point, with the $'f'$ indicating a floating point number.
- Note that the number is rounded if there are more numbers after the decimal point than you specified.

Example

$'\{0 : .2f\}'.format(1.2655)$
$'1.27'$

- If we use $.2f$ and the number you are printing is an integer, then zeros are added at the end:

Example

$'\{0\}'.format(1)$
$'1.00'$

Finding the Roots of a Quadratic Eqn

- Solving linear equation

Example

x $=$ 10 - 500 $+$79

x

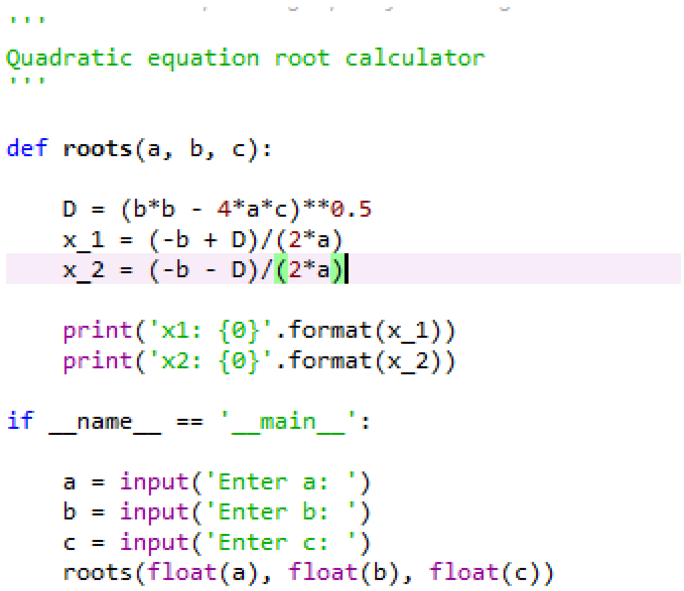
-411

- Let the generic quadratic equation be $ax^2 + bx + c$
- Let the roots of the generic equation be $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$ and $x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$
- Comparing the equation $x^2 + 2x + 1 = 0$ to the generic quadratic equation, $ax^2 + bx + c$ we see that $a = 1, b = 2$, and $c = 1$.
- We can substitute these values directly into the quadratic formula to calculate the value of x_1 and x_2.
- In Python, we first store the values of a, b, and c as the labels a, b, and c with the appropriate values:

Example

```
a = 1
b = 2
c = 1
```

- In both x_1 and x_2 we have $\frac{\sqrt{b^2-4ac}}{2a}$
- We define $\frac{\sqrt{b^2-4ac}}{2a}$ with a new label $D = \frac{\sqrt{b^2-4ac}}{2a}$
- $D = (b**2 - 4*a*c)**0.5$



```python
'''
Quadratic equation root calculator
'''


def roots(a, b, c):

    D = (b*b - 4*a*c)**0.5
    x_1 = (-b + D)/(2*a)
    x_2 = (-b - D)/(2*a)

    print('x1: {0}'.format(x_1))
    print('x2: {0}'.format(x_2))

if __name__ == '__main__':

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

Converting Units of Measurement

- The International System of Units defines seven base quantities.
- Length, mass, time and temperature are four of the seven basic quantities.
- Standard unit of measurements: meter (Length), second(time), kilogram(mass), and kelvin(temperature), respectively.
- An inch is equal to 2.54 centimeters.
- We can use the multiplication operation to convert a measurement in inches to centimeters. We can then divide the measurement in centimeters by 100 to obtain the measurement in meters.

- For example, heres how we can convert 30.5 inches to meters:

Example

```
(30.5 * 2.54) / 100
Out put: 0.7746999999999999
```

- A mile is roughly equivalent to 1.609 kilometers. If our destination is 650 miles away, then we're $650 * 1.609$ kilometers away:

Example

```
650 * 1.609
1045.85
```

- **Temperature conversion**
- Temperature expressed in Fahrenheit is converted into its equivalent value in Celsius using the formula
- $C = (F - 32) \times \frac{5}{9}$
- Where F is the temperature in Fahrenheit and C is its equivalent in Celsius.
- 98.6 degrees Fahrenheit is said to be the normal human body temperature.
- To find the corresponding temperature in degrees Celsius, we evaluate the above formula in Python:

Example

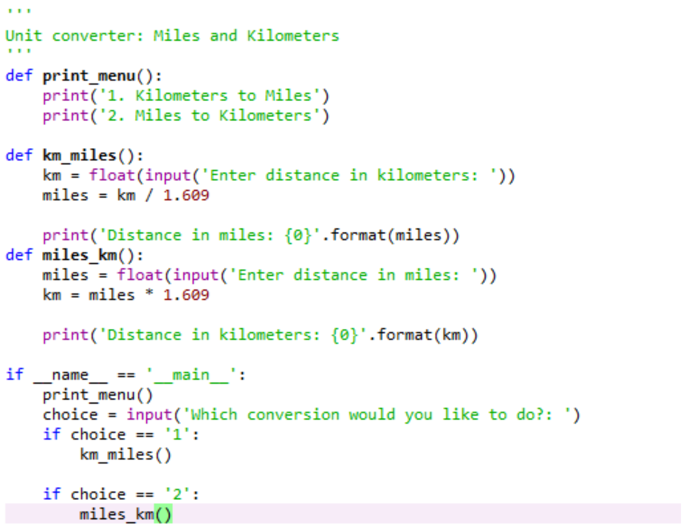
$F = 98.6$
$(F - 32) * (5/9)$
Out put: 37.0

- To convert temperature from Celsius to Fahrenheit, use the formula
- $F = (C \times \frac{9}{5}) + 32.$

Example

$C = 37$
$C * (9/5) + 32$
Out put: 98.60000000000001

- We create a label, C, with the value 37 (the normal human body temperature in Celsius). Then, we convert it into Fahrenheit using the formula, and the result is 98.6 degrees.



```python
'''
Unit converter: Miles and Kilometers
'''
def print_menu():
    print('1. Kilometers to Miles')
    print('2. Miles to Kilometers')

def km_miles():
    km = float(input('Enter distance in kilometers: '))
    miles = km / 1.609

    print('Distance in miles: {0}'.format(miles))
def miles_km():
    miles = float(input('Enter distance in miles: '))
    km = miles * 1.609

    print('Distance in kilometers: {0}'.format(km))

if __name__ == '__main__':
    print_menu()
    choice = input('Which conversion would you like to do?: ')
    if choice == '1':
        km_miles()

    if choice == '2':
        miles_km()
```

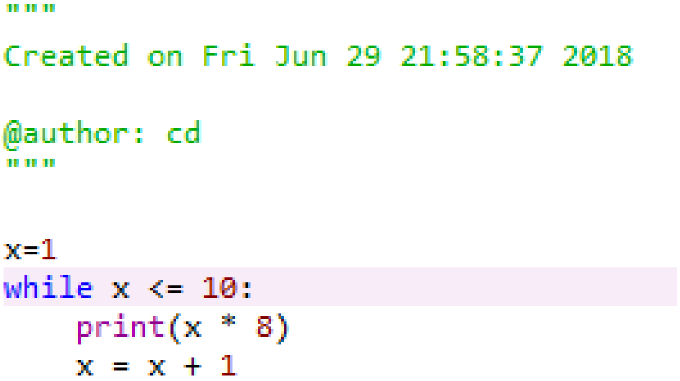
- **Programming Challenges**
- The unit conversion program we wrote is limited to conversions between kilometers and miles. Extend the program to convert between units of mass (such as kilograms and pounds) and between units of temperature (such as Celsius and Fahrenheit).
- **Even-Odd Vending Machine**
- Try writing an "even-odd vending machine," which will take a number as input and do two things:

  1. Print whether the number is even or odd.
  2. Display the number followed by the next 9 even or odd numbers.

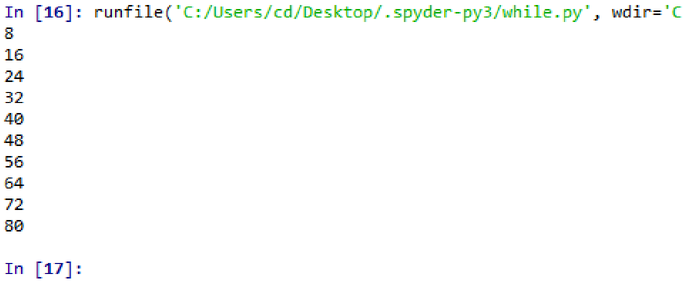
  If the input is 2, the program should print even and then print 2, 4, 6, 8, 10, 12, 14, 16, 18, 20. Similarly, if the input is 1, the program should print odd and then print 1, 3, 5, 7, 9, 11, 13, 15, 17, 19.

While loop

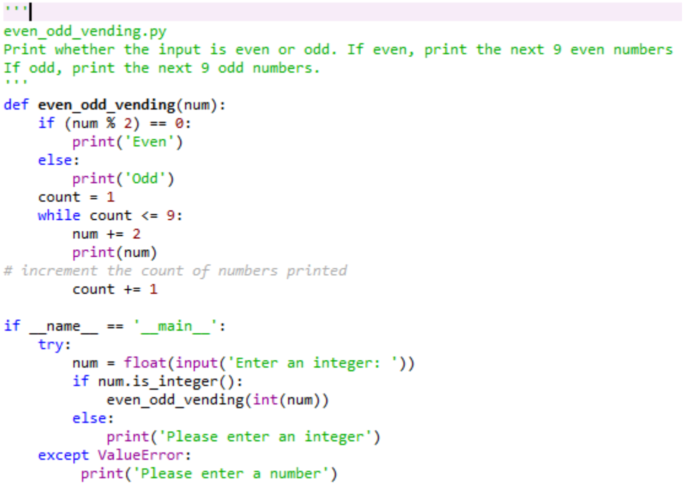
- The following program defines a variable x and assigns it an initial value of 1. Inside the while loop $x * 8$ is printed and the value of x is incremented.



```python
"""

Created on Fri Jun 29 21:58:37 2018

@author: cd
"""


x=1
while x <= 10:
    print(x * 8)
    x = x + 1
```

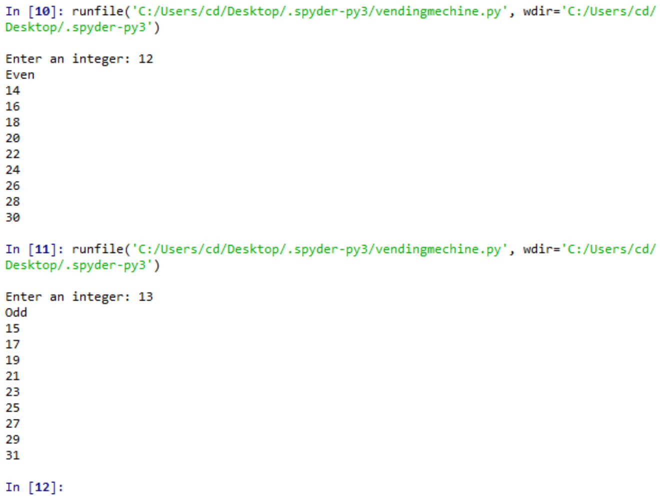


```
In [16]: runfile('C:/Users/cd/Desktop/.spyder-py3/while.py', wdir='C
8
16
24
32
40
48
56
64
72
80

In [17]:
```



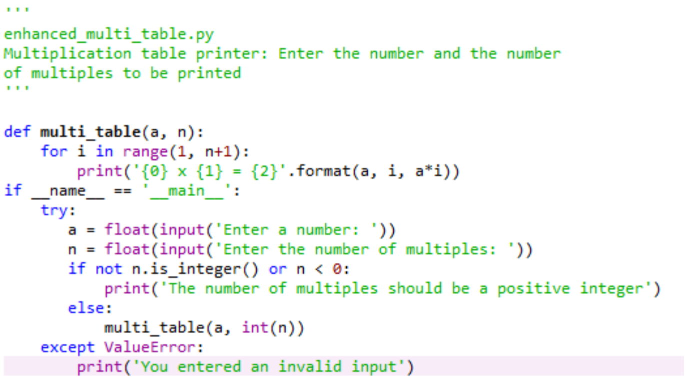
```python
'''
even_odd_vending.py
Print whether the input is even or odd. If even, print the next 9 even numbers
If odd, print the next 9 odd numbers.
'''
def even_odd_vending(num):
    if (num % 2) == 0:
        print('Even')
    else:
        print('Odd')
    count = 1
    while count <= 9:
        num += 2
        print(num)
# increment the count of numbers printed
        count += 1

if __name__ == '__main__':
    try:
        num = float(input('Enter an integer: '))
        if num.is_integer():
            even_odd_vending(int(num))
        else:
            print('Please enter an integer')
    except ValueError:
         print('Please enter a number')
```

Output:



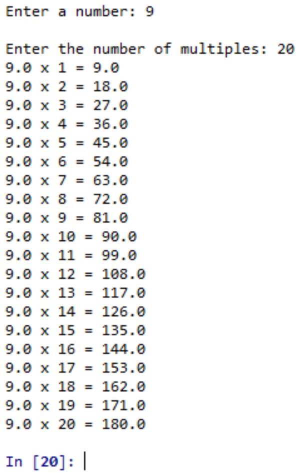
```
In [10]: runfile('C:/Users/cd/Desktop/.spyder-py3/vendingmechine.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')

Enter an integer: 12
Even
14
16
18
20
22
24
26
28
30

In [11]: runfile('C:/Users/cd/Desktop/.spyder-py3/vendingmechine.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')

Enter an integer: 13
Odd
15
17
19
21
23
25
27
29
31

In [12]:
```

- **Enhanced Multiplication Table Generator**
- Enhance the multiplication table generator so that the user can specify both the number and up to which multiple. For example, we should be able to input that we want to see a table listing the first 20 multiples of 9.



```python
'''
enhanced_multi_table.py
Multiplication table printer: Enter the number and the number
of multiples to be printed
'''

def multi_table(a, n):
    for i in range(1, n+1):
        print('{0} x {1} = {2}'.format(a, i, a*i))
if __name__ == '__main__':
    try:
        a = float(input('Enter a number: '))
        n = float(input('Enter the number of multiples: '))
        if not n.is_integer() or n < 0:
            print('The number of multiples should be a positive integer')
        else:
            multi_table(a, int(n))
    except ValueError:
        print('You entered an invalid input')
```

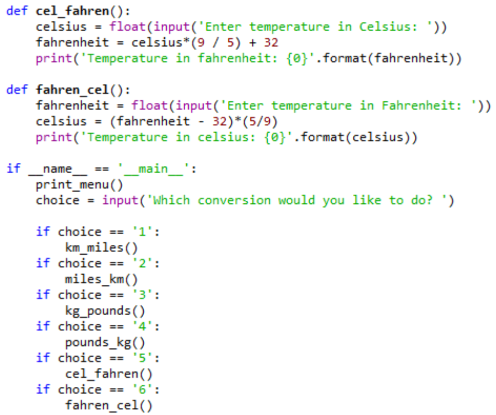
Output



```
Enter a number: 9

Enter the number of multiples: 20
9.0 x 1 = 9.0
9.0 x 2 = 18.0
9.0 x 3 = 27.0
9.0 x 4 = 36.0
9.0 x 5 = 45.0
9.0 x 6 = 54.0
9.0 x 7 = 63.0
9.0 x 8 = 72.0
9.0 x 9 = 81.0
9.0 x 10 = 90.0
9.0 x 11 = 99.0
9.0 x 12 = 108.0
9.0 x 13 = 117.0
9.0 x 14 = 126.0
9.0 x 15 = 135.0
9.0 x 16 = 144.0
9.0 x 17 = 153.0
9.0 x 18 = 162.0
9.0 x 19 = 171.0
9.0 x 20 = 180.0

In [20]:
```

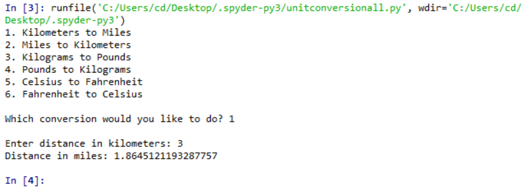
- **Enhanced Unit Converter**
- Try extending the program to convert between units of mass (such as kilograms and pounds) and between units of temperature (such as Celsius and Fahrenheit).

```
'''
enhanced_unit_converter.py
Unit converter:
Kilometers and Miles
Kilograms and Pounds
Celsius and Fahrenheit
'''
def print_menu():
    print('1. Kilometers to Miles')
    print('2. Miles to Kilometers')
    print('3. Kilograms to Pounds')
    print('4. Pounds to Kilograms')
    print('5. Celsius to Fahrenheit')
    print('6. Fahrenheit to Celsius')
def km_miles():

    km = float(input('Enter distance in kilometers: '))
    miles = km / 1.609
    print('Distance in miles: {0}'.format(miles))

def miles_km():
    miles = float(input('Enter distance in miles: '))
    km = miles * 1.609
    print('Distance in kilometers: {0}'.format(km))

def kg_pounds():
    kg = float(input('Enter weight in kilograms: '))
    pounds = kg * 2.205
    print('Weight in pounds: {0}'.format(pounds))

def pounds_kg():
    pounds = float(input('Enter weight in pounds: '))
    kg = pounds / 2.205
    print('Weight in kilograms: {0}'.format(kg))
```



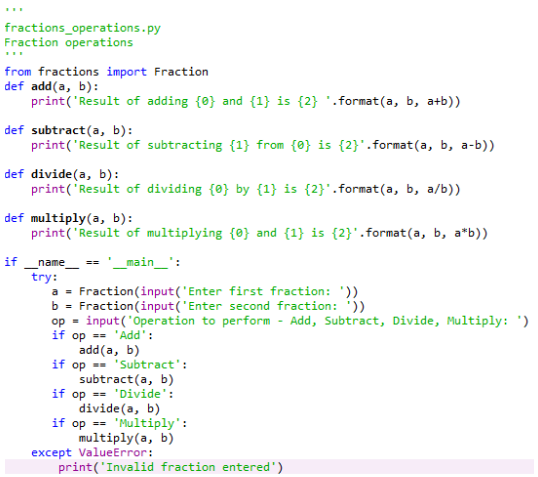
```python
def cel_fahren():
    celsius = float(input('Enter temperature in Celsius: '))
    fahrenheit = celsius*(9 / 5) + 32
    print('Temperature in fahrenheit: {0}'.format(fahrenheit))

def fahren_cel():
    fahrenheit = float(input('Enter temperature in Fahrenheit: '))
    celsius = (fahrenheit - 32)*(5/9)
    print('Temperature in celsius: {0}'.format(celsius))

if __name__ == '__main__':
    print_menu()
    choice = input('Which conversion would you like to do? ')

    if choice == '1':
        km_miles()
    if choice == '2':
        miles_km()
    if choice == '3':
        kg_pounds()
    if choice == '4':
        pounds_kg()
    if choice == '5':
        cel_fahren()
    if choice == '6':
        fahren_cel()
```

Output



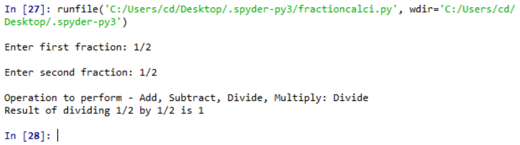
```
In [3]: runfile('C:/Users/cd/Desktop/.spyder-py3/unitconversionall.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
1. Kilometers to Miles
2. Miles to Kilometers
3. Kilograms to Pounds
4. Pounds to Kilograms
5. Celsius to Fahrenheit
6. Fahrenheit to Celsius

Which conversion would you like to do? 1

Enter distance in kilometers: 3
Distance in miles: 1.8645121193287757

In [4]:
```

- Write a calculator that can perform the basic mathematical operations on two fractions. It should ask the user for two fractions and the operation the user wants to carry out.



```python
'''
fractions_operations.py
Fraction operations
'''
from fractions import Fraction
def add(a, b):
    print('Result of adding {0} and {1} is {2} '.format(a, b, a+b))

def subtract(a, b):
    print('Result of subtracting {1} from {0} is {2}'.format(a, b, a-b))

def divide(a, b):
    print('Result of dividing {0} by {1} is {2}'.format(a, b, a/b))

def multiply(a, b):
    print('Result of multiplying {0} and {1} is {2}'.format(a, b, a*b))

if __name__ == '__main__':
    try:
       a = Fraction(input('Enter first fraction: '))
       b = Fraction(input('Enter second fraction: '))
       op = input('Operation to perform - Add, Subtract, Divide, Multiply: ')
       if op == 'Add':
           add(a, b)
       if op == 'Subtract':
           subtract(a, b)
       if op == 'Divide':
           divide(a, b)
       if op == 'Multiply':
           multiply(a, b)
    except ValueError:
        print('Invalid fraction entered')
```

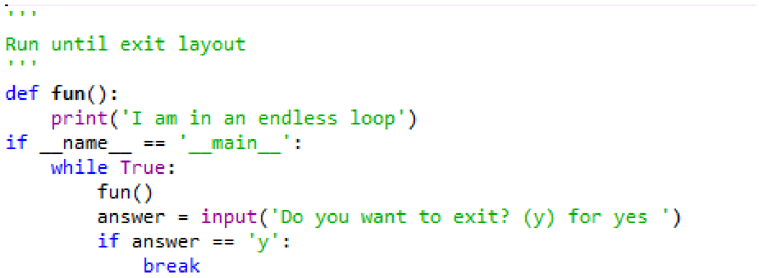
Output



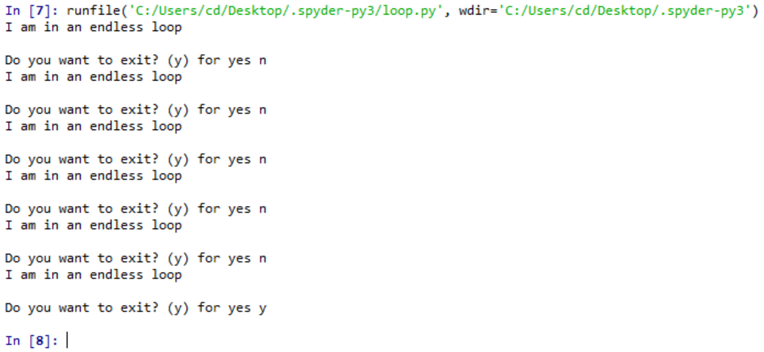
```
In [27]: runfile('C:/Users/cd/Desktop/.spyder-py3/fractioncalci.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')

Enter first fraction: 1/2

Enter second fraction: 1/2

Operation to perform - Add, Subtract, Divide, Multiply: Divide
Result of dividing 1/2 by 1/2 is 1

In [28]: 
```

Give Exit Power to the User

- All the programs we have written so far work only for one iteration of input and output. For example, consider the program to print the multiplication table: the user executes the program and enters a number; then the program prints the multiplication table and exits. If the user wanted to print the multiplication table of another number, the program would have to be rerun.
- It would be more convenient if the user could choose whether to exit or continue using the program. The key to writing such programs is to set up an infinite loop, or a loop that doesnt exit unless explicitly asked to do so.

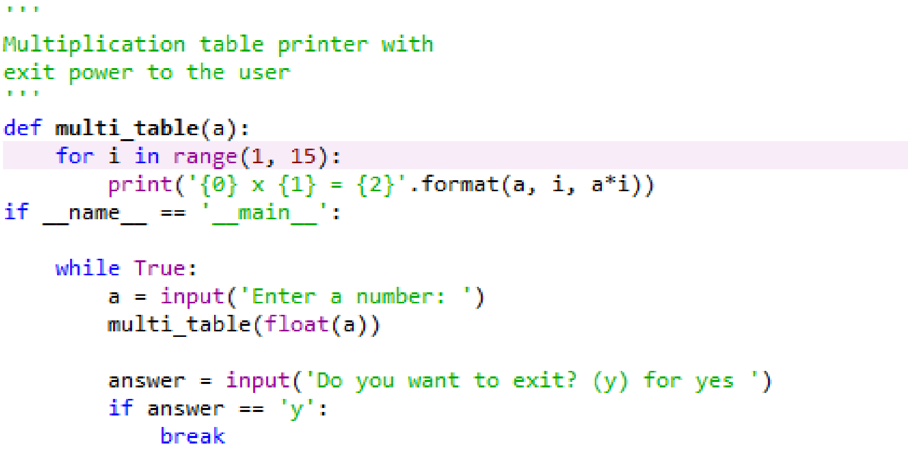


```
'''
Run until exit layout
'''
def fun():
    print('I am in an endless loop')
if __name__ == '__main__':
    while True:
        fun()
        answer = input('Do you want to exit? (y) for yes ')
        if answer == 'y':
            break
```

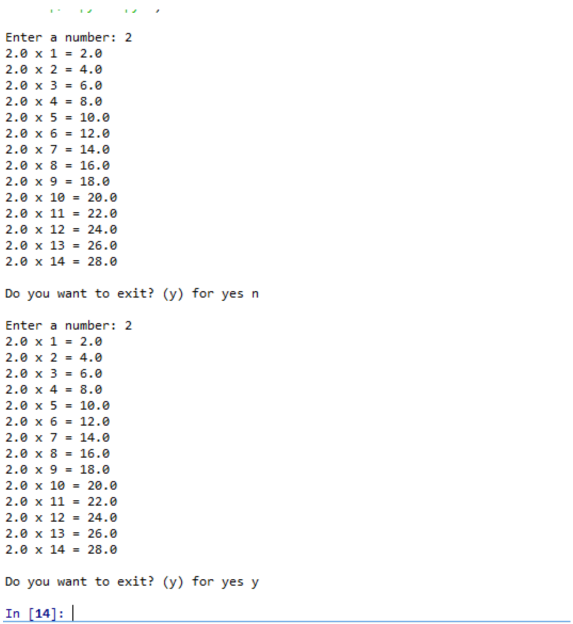


```
In [7]: runfile('C:/Users/cd/Desktop/.spyder-py3/loop.py', wdir='C:/Users/cd/Desktop/.spyder-py3')
I am in an endless loop

Do you want to exit? (y) for yes n
I am in an endless loop

Do you want to exit? (y) for yes n
I am in an endless loop

Do you want to exit? (y) for yes n
I am in an endless loop

Do you want to exit? (y) for yes n
I am in an endless loop

Do you want to exit? (y) for yes n
I am in an endless loop

Do you want to exit? (y) for yes y

In [8]:
```

- We rewrite the multiplication table generator so that it keeps going until the user wants to exit. The new version of the program is shown below:



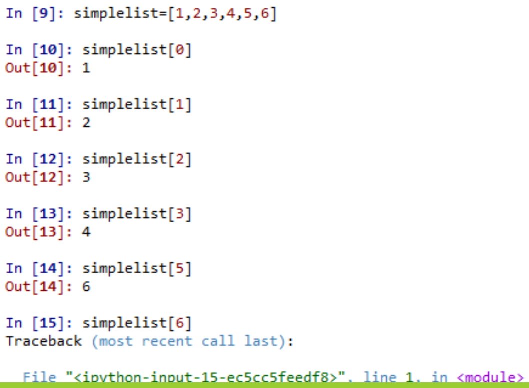
```
'''
Multiplication table printer with
exit power to the user
'''
def multi_table(a):
    for i in range(1, 15):
        print('{0} x {1} = {2}'.format(a, i, a*i))
if __name__ == '__main__':

    while True:
        a = input('Enter a number: ')
        multi_table(float(a))

        answer = input('Do you want to exit? (y) for yes ')
        if answer == 'y':
            break
```



```
Enter a number: 2
2.0 x 1 = 2.0
2.0 x 2 = 4.0
2.0 x 3 = 6.0
2.0 x 4 = 8.0
2.0 x 5 = 10.0
2.0 x 6 = 12.0
2.0 x 7 = 14.0
2.0 x 8 = 16.0
2.0 x 9 = 18.0
2.0 x 10 = 20.0
2.0 x 11 = 22.0
2.0 x 12 = 24.0
2.0 x 13 = 26.0
2.0 x 14 = 28.0

Do you want to exit? (y) for yes n

Enter a number: 2
2.0 x 1 = 2.0
2.0 x 2 = 4.0
2.0 x 3 = 6.0
2.0 x 4 = 8.0
2.0 x 5 = 10.0
2.0 x 6 = 12.0
2.0 x 7 = 14.0
2.0 x 8 = 16.0
2.0 x 9 = 18.0
2.0 x 10 = 20.0
2.0 x 11 = 22.0
2.0 x 12 = 24.0
2.0 x 13 = 26.0
2.0 x 14 = 28.0

Do you want to exit? (y) for yes y

In [14]:
```

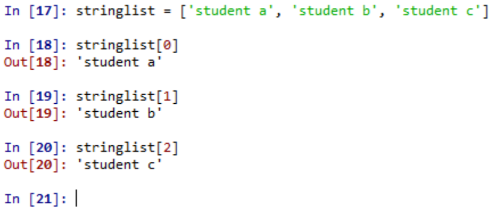
Visualizing Data with Graphs.

- **Visualizing Data with Graphs.**
- We can create a list by entering values, separated by commas, between square brackets.
- The following statement creates a list and uses the label simplelist



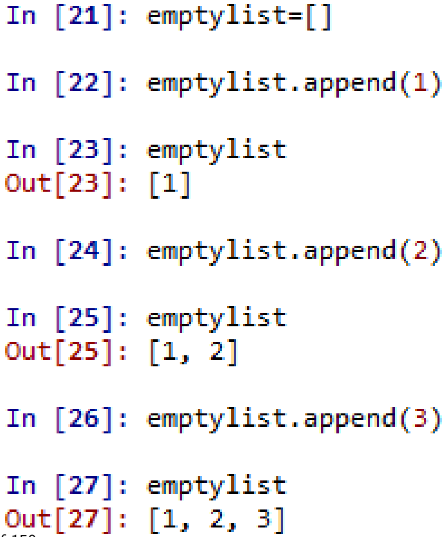
```
In [9]: simplelist=[1,2,3,4,5,6]

In [10]: simplelist[0]
Out[10]: 1

In [11]: simplelist[1]
Out[11]: 2

In [12]: simplelist[2]
Out[12]: 3

In [13]: simplelist[3]
Out[13]: 4

In [14]: simplelist[5]
Out[14]: 6

In [15]: simplelist[6]
Traceback (most recent call last):

  File "<ipython-input-15-ec5cc5feedf8>", line 1, in <module>
```

- Notice that the first item of the list is at index 0, the second item is at index 1, and so on that is, the positions in the list start counting from 0, not 1.
- Lists can store strings, too:



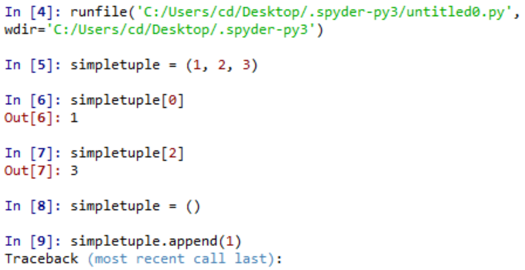
```
In [17]: stringlist = ['student a', 'student b', 'student c']

In [18]: stringlist[0]
Out[18]: 'student a'

In [19]: stringlist[1]
Out[19]: 'student b'

In [20]: stringlist[2]
Out[20]: 'student c'

In [21]: 
```

- An empty list is just that - a list with no items or elements - and it can be created like this:



```
In [21]: emptylist=[]

In [22]: emptylist.append(1)

In [23]: emptylist
Out[23]: [1]

In [24]: emptylist.append(2)

In [25]: emptylist
Out[25]: [1, 2]

In [26]: emptylist.append(3)

In [27]: emptylist
Out[27]: [1, 2, 3]
```

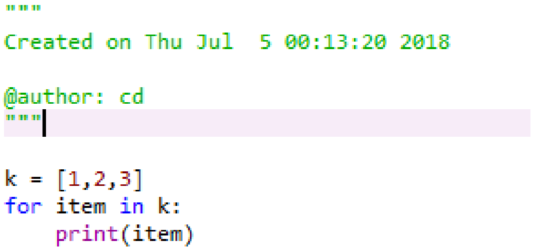
- We can create an empty list and then use the *append*() method to add items later as shown above.
- Note that when we use .*append*(), the value gets added to the end of the list. This is just one way of adding values to a list.
- Creating a tuple is similar to creating a list, but instead of square brackets, we use parentheses:
- However, there is no append() method to add a new value to an existing tuple, so we can not add values to an empty tuple.


```
In [4]: runfile('C:/Users/cd/Desktop/.spyder-py3/untitled0.py',
wdir='C:/Users/cd/Desktop/.spyder-py3')

In [5]: simpletuple = (1, 2, 3)

In [6]: simpletuple[0]
Out[6]: 1

In [7]: simpletuple[2]
Out[7]: 3

In [8]: simpletuple = ()

In [9]: simpletuple.append(1)
Traceback (most recent call last):
```

- Once we create a tuple, then contents of the tuple can't be changed.

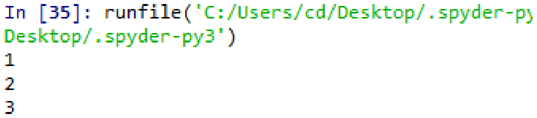
Iterating over a List or Tuple

- We can go over a list or tuple using a for loop as follows:



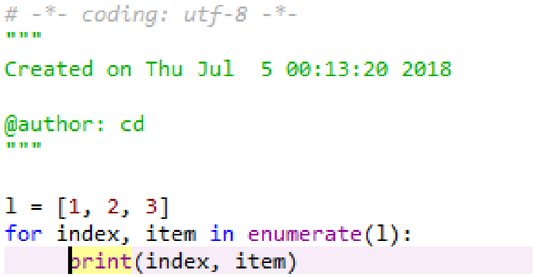
```
"""
Created on Thu Jul  5 00:13:20 2018

@author: cd
"""

k = [1,2,3]
for item in k:
    print(item)
```

- This will print the items in the list:

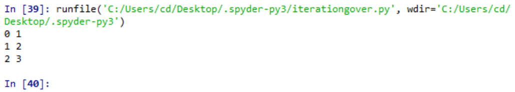


```
In [35]: runfile('C:/Users/cd/Desktop/.spyder-py
Desktop/.spyder-py3')
1
2
3
```

- Sometimes we might need to know the position or the index of an item in a list or tuple. we can use the *enumerate*() function to iterate over all the items of a list and return the index of an item as well as the item itself.



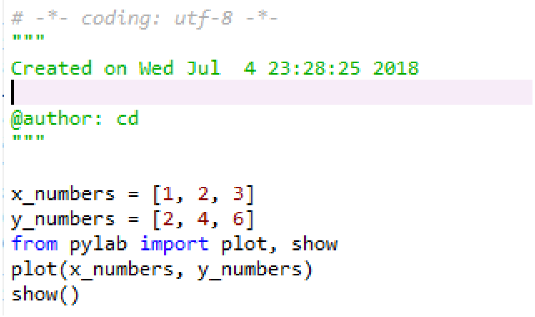
```python
# -*- coding: utf-8 -*-
"""
Created on Thu Jul  5 00:13:20 2018

@author: cd
"""


l = [1, 2, 3]
for index, item in enumerate(l):
    print(index, item)
```



```
In [39]: runfile('C:/Users/cd/Desktop/.spyder-py3/iterationgover.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
0 1
1 2
2 3

In [40]:
```

- **Creating Graphs with Matplotlib**
- Matplotlib is a Python package, which means that it is a collection of modules with related functionality.
- We will start with a simple graph with just three points: $(1,2),(2,4)$, and $(3,6)$.
- To create this graph, we will first make two lists of numbers one storing the values of the x-coordinates of these points and another storing the y-coordinates.

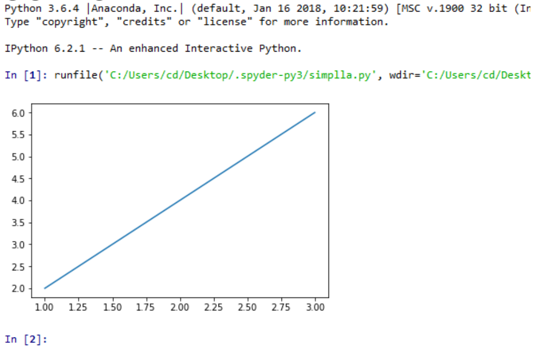
- In the first line, we import the *plot*() and *show*() functions from the pylab module, which is part of the Matplotlib package.



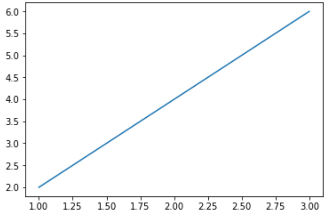
```
# -*- coding: utf-8 -*-
"""
Created on Wed Jul  4 23:28:25 2018

@author: cd
"""


x_numbers = [1, 2, 3]
y_numbers = [2, 4, 6]
from pylab import plot, show
plot(x_numbers, y_numbers)
show()
```

- The plot() function only creates the graph. To actually display it, we have to call the show() function:

OUTPUT



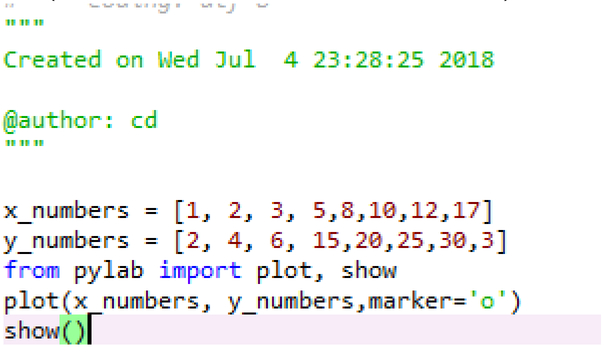
```
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:21:59) [MSC v.1900 32 bit (In
Type "copyright", "credits" or "license" for more information.

IPython 6.2.1 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/cd/Desktop/.spyder-py3/simplla.py', wdir='C:/Users/cd/Deskt
```



```
In [2]:
```

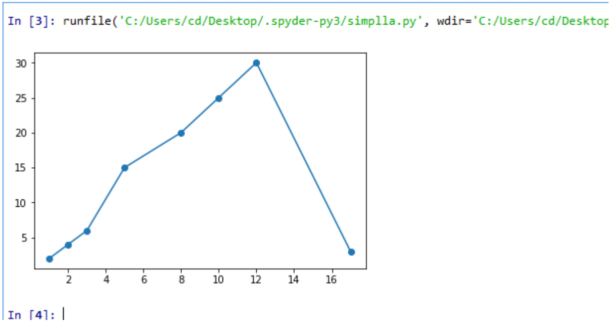
Marking Points on Graph

- To mark the points we can use an additional keyword argument while calling the *plot*() function:
- $plot(x\_numbers, y\_numbers, marker =' o')$



```
"""
Created on Wed Jul  4 23:28:25 2018

@author: cd
"""


x_numbers = [1, 2, 3, 5,8,10,12,17]
y_numbers = [2, 4, 6, 15,20,25,30,3]
from pylab import plot, show
plot(x_numbers, y_numbers,marker='o')
show()
```

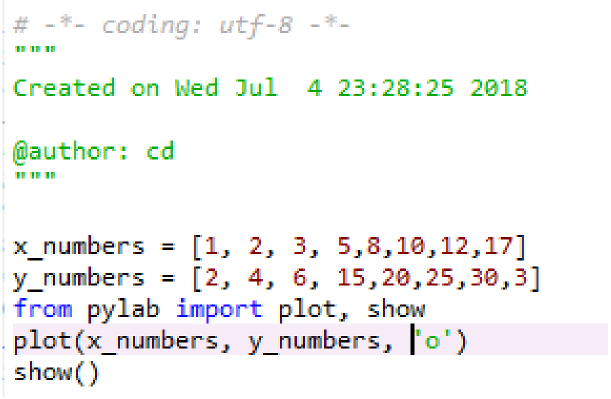
- By entering marker='o', we tell Python to mark each point from our lists with a small dot that looks like an o. Once you enter show() again, we see that each point is marked with a dot



```
In [3]: runfile('C:/Users/cd/Desktop/.spyder-py3/simplla.py', wdir='C:/Users/cd/Desktop
```

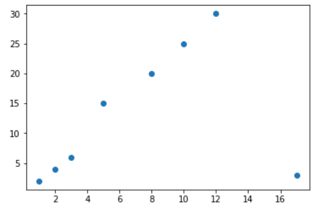
```
In [4]:
```

- Marker options include $'o', '*', 'x', and' +'$.
- We can also make a graph that marks only the points that you specified, without any line connecting them, by omitting $marker =$
- $plot(x\_numbers, y\_numbers, 'o')$



```
# -*- coding: utf-8 -*-
"""

Created on Wed Jul  4 23:28:25 2018

@author: cd
"""


x_numbers = [1, 2, 3, 5,8,10,12,17]
y_numbers = [2, 4, 6, 15,20,25,30,3]
from pylab import plot, show
plot(x_numbers, y_numbers, 'o')
show()
```

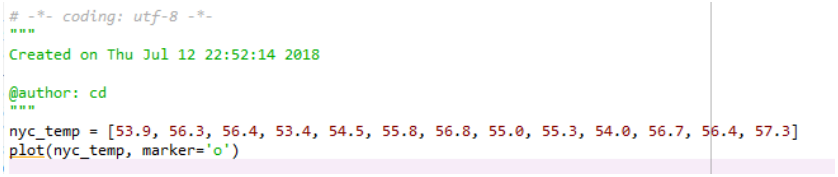
```
In [4]: runfile('C:/Users/cd/Desktop/.spyder-py3/simplla.py', wdir='C:/Users/cd/Desktop/.spyder-py3')
```



```
In [5]: |
```

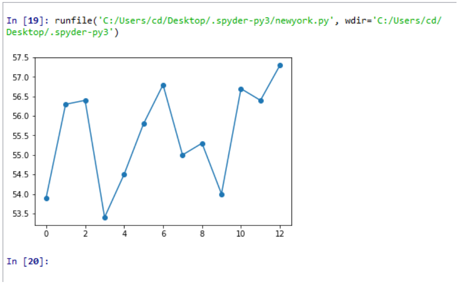
Graphing the Average Annual Temperature in New York City

- The average annual temperatures for New York City measured at Central Park, specifically-during the years 2000 to 2012 are as follows: 53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, and 57.3 degrees Fahrenheit.



```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 12 22:52:14 2018

@author: cd
"""
nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
plot(nyc_temp, marker='o')
```

Out Put

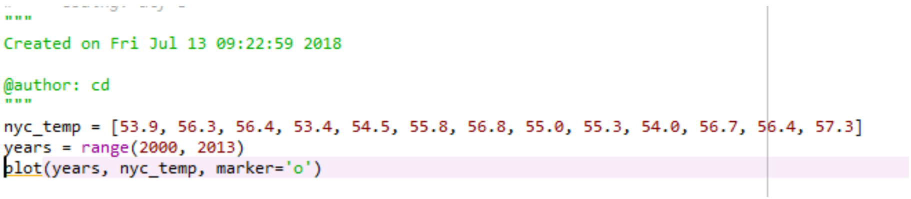


```
In [19]: runfile('C:/Users/cd/Desktop/.spyder-py3/newyork.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```

```
In [20]:
```

- We store the average temperatures in a list, nyc_temp. Then, we call the function plot() passing only this list (and the marker string). When we use plot() on a single list, those numbers are automatically plotted on the y-axis. The corresponding values on the x-axis are filled in as the positions of each value in the list.
- That is, the first temperature value, 53.9, gets a corresponding x-axis value of 0 because it's in position 0 of the list (remember, the list position starts counting from 0, not 1).

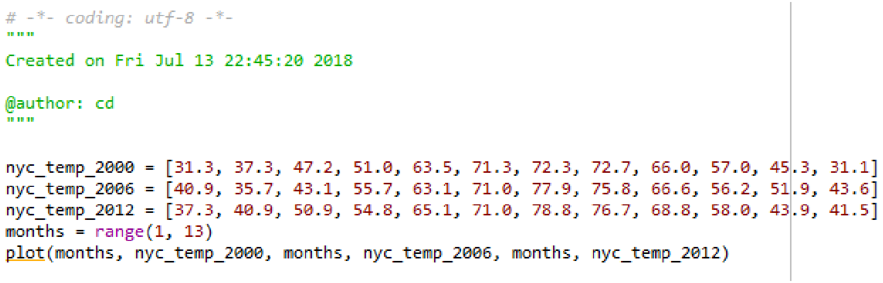
- We can specify the years on the X-Axis
- We can easily do this by creating another list with the years in it and then calling the plot() function:



```
"""
Created on Fri Jul 13 09:22:59 2018

@author: cd
"""

nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
years = range(2000, 2013)
plot(years, nyc_temp, marker='o')
```

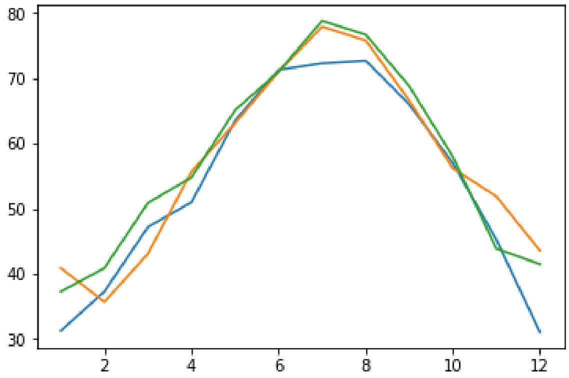
Comparing the Monthly Temperature Trends of New York City

- While still looking at New York City, let's see how the average monthly temperature has varied over the years.
- This will give us a chance to understand how to plot multiple lines on a single graph.
- We'll choose three years: 2000, 2006, and 2012. For each of these years, we'll plot the average temperature for all 12 months.
- First, we need to create three lists to store the temperature (in Fahrenheit). Each list will consist of 12 numbers corresponding to the average temperature from January to December each year:



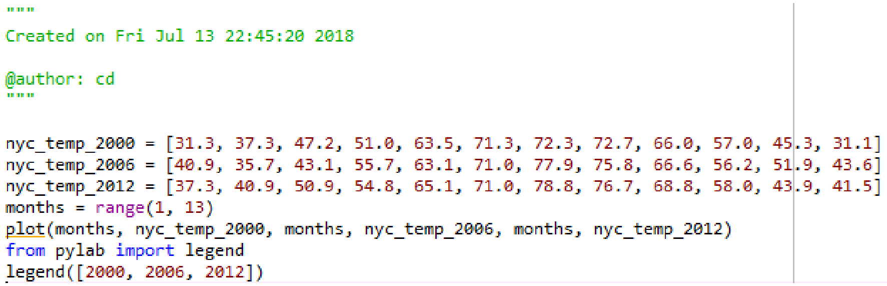
```
# -*- coding: utf-8 -*-
"""
Created on Fri Jul 13 22:45:20 2018

@author: cd
"""

nyc_temp_2000 = [31.3, 37.3, 47.2, 51.0, 63.5, 71.3, 72.3, 72.7, 66.0, 57.0, 45.3, 31.1]
nyc_temp_2006 = [40.9, 35.7, 43.1, 55.7, 63.1, 71.0, 77.9, 75.8, 66.6, 56.2, 51.9, 43.6]
nyc_temp_2012 = [37.3, 40.9, 50.9, 54.8, 65.1, 71.0, 78.8, 76.7, 68.8, 58.0, 43.9, 41.5]
months = range(1, 13)
plot(months, nyc_temp_2000, months, nyc_temp_2006, months, nyc_temp_2012)
```

```
In [12]: runfile('C:/Users/cd/Desktop/.spyder-py3/months.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```



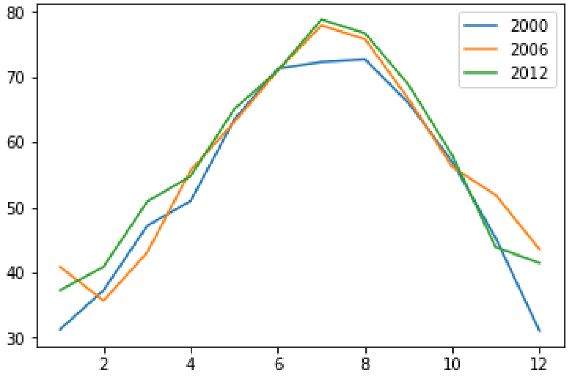
```
In [13]:
```

- Now we have three plots all on one graph. Python automatically chooses a different color for each line to indicate that the lines have been plotted from different data sets.
- We have a problem, however, because we don't have any clue as to which color corresponds to which year. To fix this, we can use the function legend(), which lets us add a legend to the graph.
- A legend is a small display box that identifies what different parts of the graph mean.
- Here, we'll use a legend to indicate which year each colored line stands for.



```
"""
Created on Fri Jul 13 22:45:20 2018

@author: cd
"""

nyc_temp_2000 = [31.3, 37.3, 47.2, 51.0, 63.5, 71.3, 72.3, 72.7, 66.0, 57.0, 45.3, 31.1]
nyc_temp_2006 = [40.9, 35.7, 43.1, 55.7, 63.1, 71.0, 77.9, 75.8, 66.6, 56.2, 51.9, 43.6]
nyc_temp_2012 = [37.3, 40.9, 50.9, 54.8, 65.1, 71.0, 78.8, 76.7, 68.8, 58.0, 43.9, 41.5]
months = range(1, 13)
plot(months, nyc_temp_2000, months, nyc_temp_2006, months, nyc_temp_2012)
from pylab import legend
legend([2000, 2006, 2012])
```

```
In [13]: runfile('C:/Users/cd/Desktop/.spyder-py3/months.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```

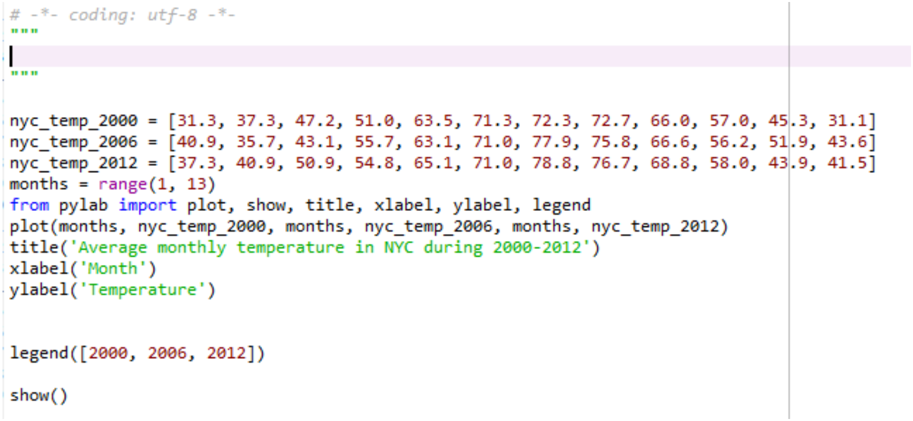


```
In [14]:
```

- We can specify a particular position for display box, use command $legend([2000, 2006, 2012], bbox\_to\_anchor = (.75, .85))$

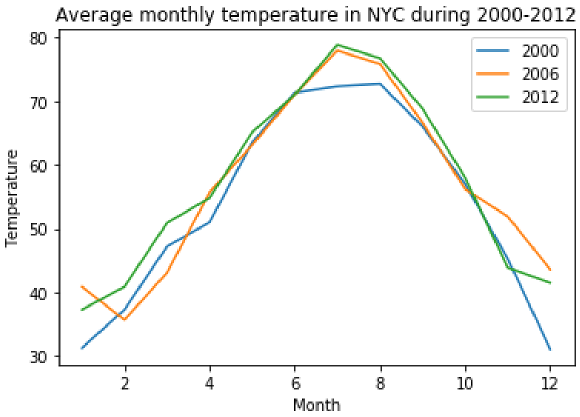
Customizing Graphs

- We can add a title to our graph using the title() function and add labels for the x-axes and y-axes using the xlabel() and ylabel() functions.
- Let us re-create the last plot and add all this additional information:



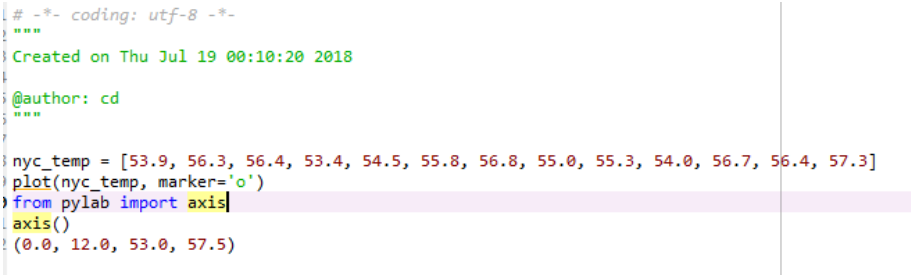
```
# -*- coding: utf-8 -*-
"""

"""

nyc_temp_2000 = [31.3, 37.3, 47.2, 51.0, 63.5, 71.3, 72.3, 72.7, 66.0, 57.0, 45.3, 31.1]
nyc_temp_2006 = [40.9, 35.7, 43.1, 55.7, 63.1, 71.0, 77.9, 75.8, 66.6, 56.2, 51.9, 43.6]
nyc_temp_2012 = [37.3, 40.9, 50.9, 54.8, 65.1, 71.0, 78.8, 76.7, 68.8, 58.0, 43.9, 41.5]
months = range(1, 13)
from pylab import plot, show, title, xlabel, ylabel, legend
plot(months, nyc_temp_2000, months, nyc_temp_2006, months, nyc_temp_2012)
title('Average monthly temperature in NYC during 2000-2012')
xlabel('Month')
ylabel('Temperature')


legend([2000, 2006, 2012])

show()
```

```
In [9]: runfile('C:/Users/cd/Desktop/.spyder-py3/months.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```



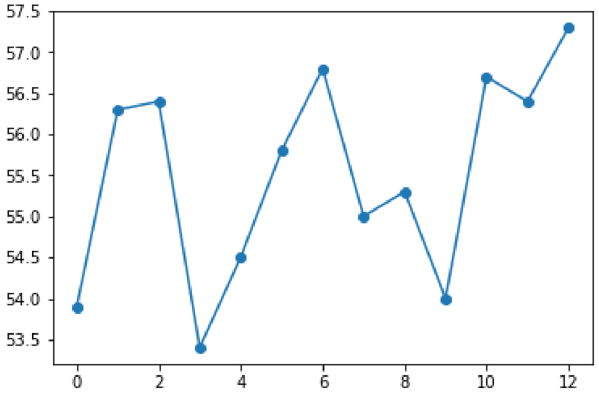
- So far, we have allowed the numbers on both axes to be automatically determined by Python based on the data supplied to the plot() function. This may be fine for most cases, but sometimes this automatic range isn't the clearest way to present the data, as we saw in the graph where we plotted the average annual temperature of New York City. There, even small changes in the temperature seemed large because the automatically chosen y-axis range was very narrow. (See NYC Temperature plot with one graph.)
- We can adjust the range of the axes using the **(**axis()) function. This function can be used both to retrieve the current range and to set a new range for the axes.

- Consider, once again, the average annual temperature of New York City during the years 2000 to 2012 and create a plot as we did earlier.
- Now, import the axis() function and call it.



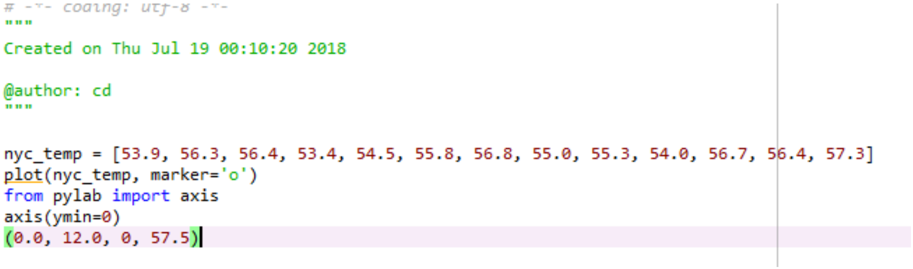
```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 19 00:10:20 2018

@author: cd
"""

nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
plot(nyc_temp, marker='o')
from pylab import axis
axis()
(0.0, 12.0, 53.0, 57.5)
```

```
In [13]: runfile('C:/Users/cd/Desktop/.spyder-py3/axis.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```



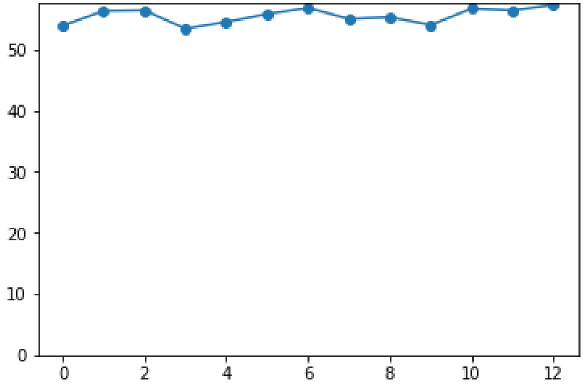
```
In [14]:
```

- The function returned a tuple with four numbers corresponding to the range for the x-axis (0.0, 12.0) and the y-axis (53.0, 57.5). These are the same range values from the graph that we made earlier. Now, let's change the y-axis to start from 0 instead of 53.0:

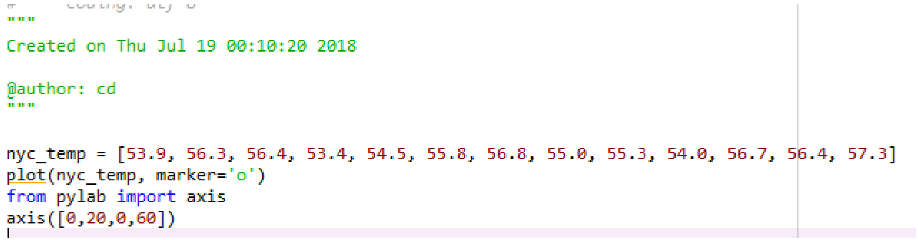


```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 19 00:10:20 2018

@author: cd
"""

nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
plot(nyc_temp, marker='o')
from pylab import axis
axis(ymin=0)
(0.0, 12.0, 0, 57.5)
```

```
In [14]: runfile('C:/Users/cd/Desktop/.spyder-py3/axis.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```

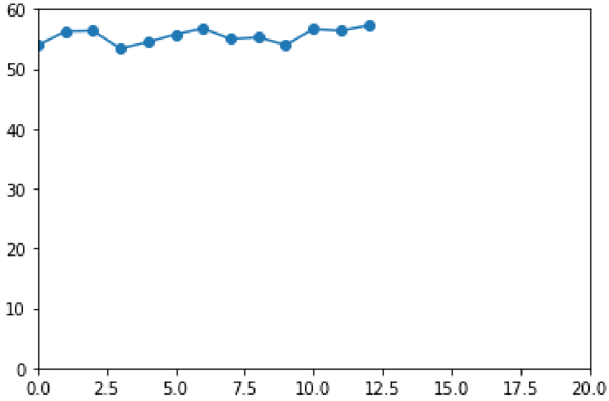


- Similarly, we can use xmin, xmax, and ymax to set the minimum and maximum values for the x-axis and the maximum value for the y-axis, respectively.
- It is easier to call the axis() function with all four range values entered as a list, such as axis([0, 20, 0, 60]).
- This would set the range of the x-axis to (0, 20) and that of the y-axis to (0, 60).



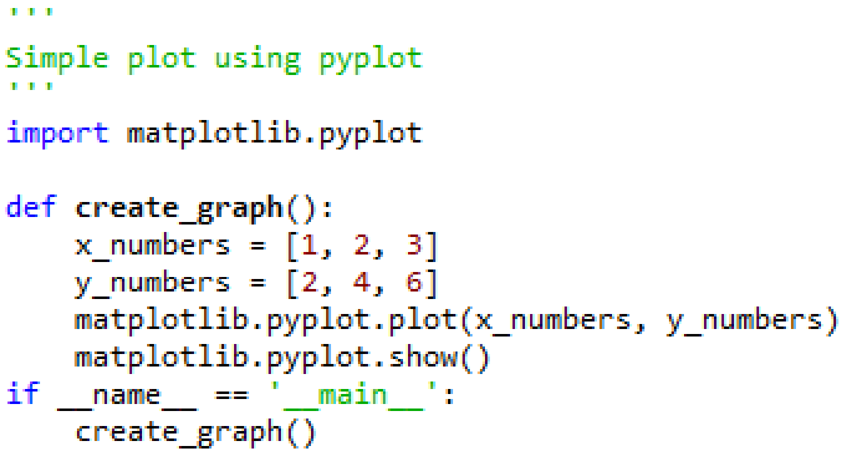
```
#      coding: utf-8
"""
Created on Thu Jul 19 00:10:20 2018

@author: cd
"""


nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
plot(nyc_temp, marker='o')
from pylab import axis
axis([0,20,0,60])
```

```
In [16]: runfile('C:/Users/cd/Desktop/.spyder-py3/axis.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```

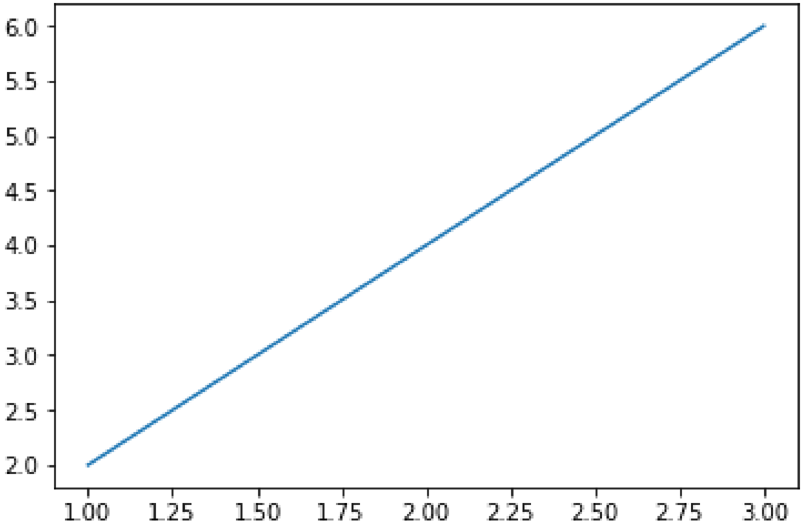


Plotting Using pyplot

- As part of a larger program- the pyplot module is more efficient than using matplotlib.
- All the methods that we learned about when using pylab will work the same way with pyplot.



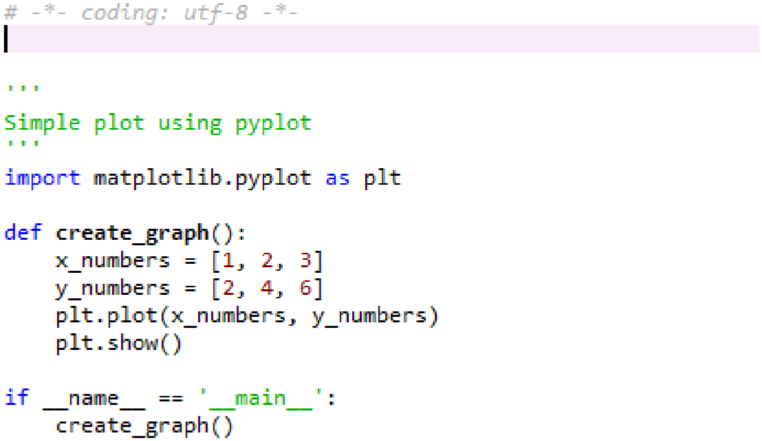
```python
'''
Simple plot using pyplot
'''
import matplotlib.pyplot

def create_graph():
    x_numbers = [1, 2, 3]
    y_numbers = [2, 4, 6]
    matplotlib.pyplot.plot(x_numbers, y_numbers)
    matplotlib.pyplot.show()
if __name__ == '__main__':
    create_graph()
```



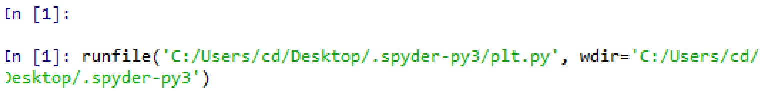
```
In [18]:
```

- Importing an entire module is useful when you're going to use a number of functions from that module. Instead of importing them individually, you can just import the whole module at once and refer to different functions when you need them.
- We call the function as matplotlib.pyplot.plot(), which means that we're calling the plot() function defined in the pyplot module of the matplotlib package.

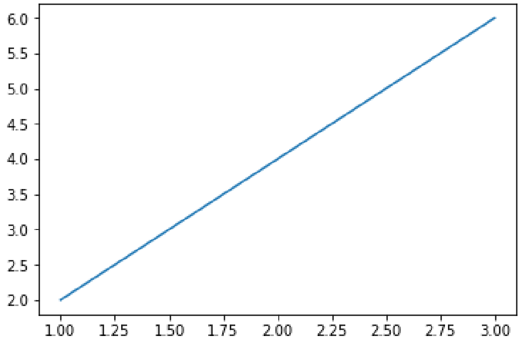
- To save us some typing, we can import the pyplot module by entering import matplotlib.pyplot as plt. Then, we can refer to pyplot with the label plt in our programs, instead of having to always type matplotlib.pyplot:



```
# -*- coding: utf-8 -*-


'''
Simple plot using pyplot
'''
import matplotlib.pyplot as plt

def create_graph():
    x_numbers = [1, 2, 3]
    y_numbers = [2, 4, 6]
    plt.plot(x_numbers, y_numbers)
    plt.show()

if __name__ == '__main__':
    create_graph()
```



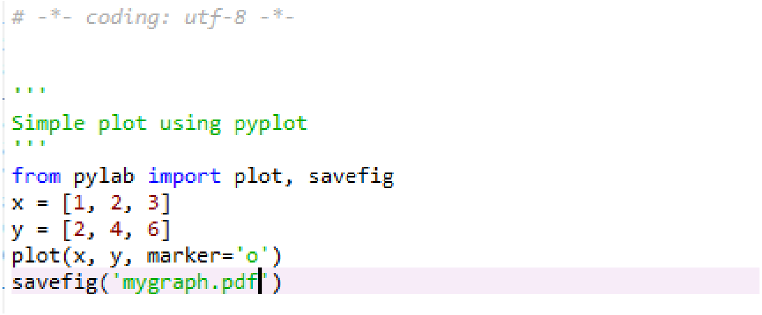
```
In [1]:

In [1]: runfile('C:/Users/cd/Desktop/.spyder-py3/plt.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```



- Now, we can call the functions by prefixing them with the shortened plt instead of matplotlib.pyplot.

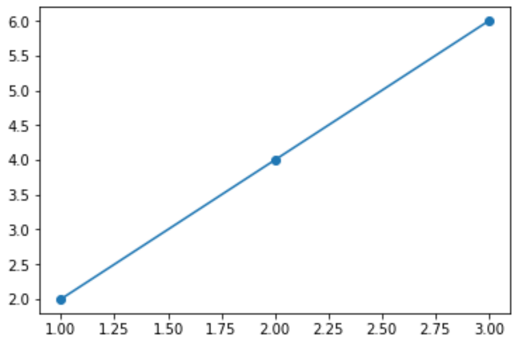
Saving the Plots

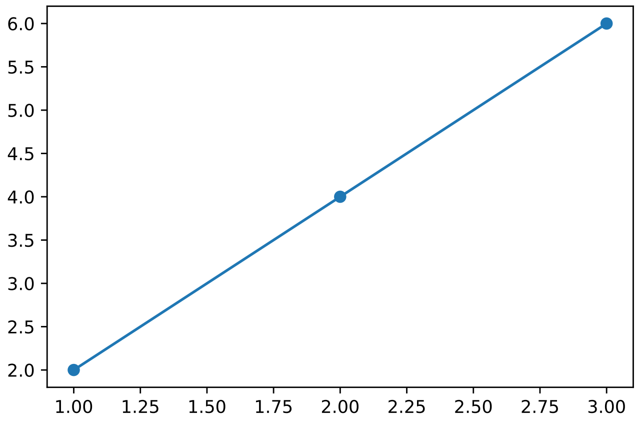
- If we need to save our graphs, we can do so using the savefig() function.
- This function saves the graph as an image file, which you can use in reports or presentations. You can choose among several image formats, including PNG, PDF, and SVG.



```
# -*- coding: utf-8 -*-


'''
Simple plot using pyplot
'''
from pylab import plot, savefig
x = [1, 2, 3]
y = [2, 4, 6]
plot(x, y, marker='o')
savefig('mygraph.pdf')
```

```
In [4]: runfile('C:/Users/cd/Desktop/.spyder-py3/plt.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```

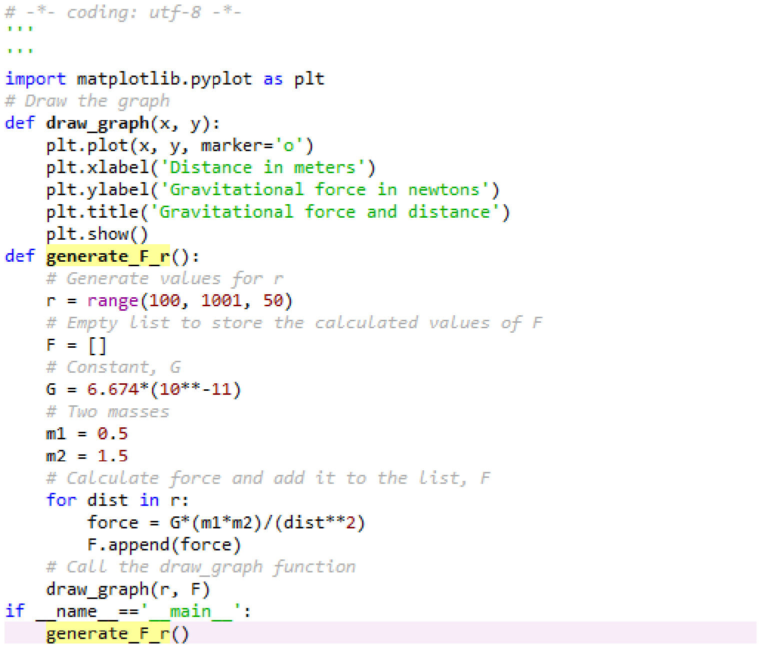



6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00

Plotting with Formulas

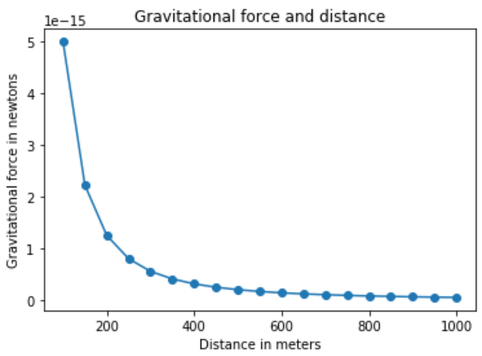
- We're going to create graphs from mathematical formulas.
  Newton's Law of Universal Gravitation
- According to Newton's law of universal gravitation, a body of mass m_1 attracts another body of mass m_2 with an amount of force F according to the formula $F = \frac{Gm_1m_2}{r^2}$, where r is the distance between the two bodies and G is the gravitational constant.
- We see what happens to the force as the distance between the two bodies increases.

- Let's take the masses of two bodies: the mass of the first body (m_1) is $0.5kg$, and the mass of the second body (m_2) is $1.5kg$.
- The value of the gravitational constant is $6.674 \times 10^{-11} Nm^2 kg^{-2}$.
- Now we're ready to calculate the gravitational force between these two bodies at 19 different distances: 100 m, 150 m, 200 m, 250 m, 300 m, and so on up through 1000 m.



```python
# -*- coding: utf-8 -*-
'''
'''
import matplotlib.pyplot as plt
# Draw the graph
def draw_graph(x, y):
    plt.plot(x, y, marker='o')
    plt.xlabel('Distance in meters')
    plt.ylabel('Gravitational force in newtons')
    plt.title('Gravitational force and distance')
    plt.show()
def generate_F_r():
    # Generate values for r
    r = range(100, 1001, 50)
    # Empty list to store the calculated values of F
    F = []
    # Constant, G
    G = 6.674*(10**-11)
    # Two masses
    m1 = 0.5
    m2 = 1.5
    # Calculate force and add it to the list, F
    for dist in r:
        force = G*(m1*m2)/(dist**2)
        F.append(force)
    # Call the draw_graph function
    draw_graph(r, F)
if __name__=='__main__':
    generate_F_r()
```

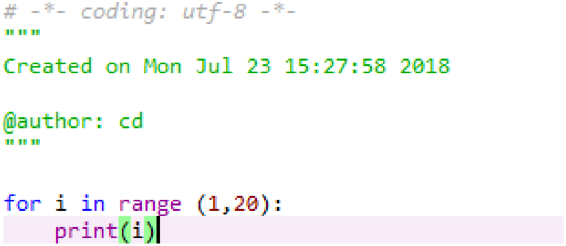
```
In [16]: runfile('C:/Users/cd/Desktop/.spyder-py3/untitled0.py', wdir='C:/Users/cd/
Desktop/.spyder-py3')
```



- As the distance (r) increases, the gravitational force decreases. With this kind of relationship, we say that the gravitational force is inversely proportional to the distance between the two bodies.
- Also, note that when the value of one of the two variables changes, the other variable won't necessarily change by the same proportion. We refer to this as a nonlinear relationship. As a result, we end up with a curved line on the graph instead of a straight one.

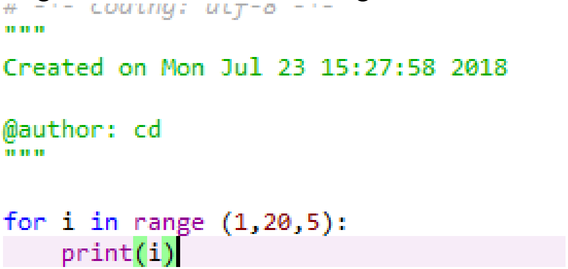
Generating Equally Spaced Floating Point Numbers

- We've used the range() function to generate equally spaced integers.



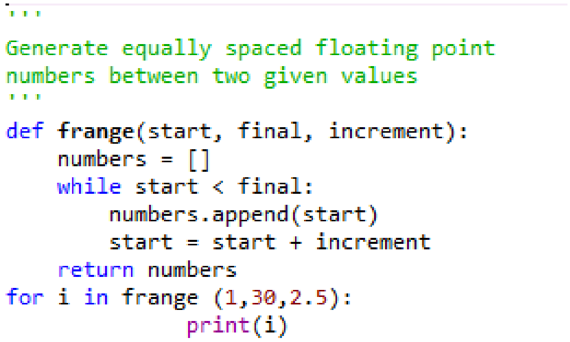
```
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 23 15:27:58 2018

@author: cd
"""

for i in range (1,20):
    print(i)
```

- If we wanted a list of integers between 1 and 10 with each integer separated by 1, we can use range(1, 10).
- If we wanted a different step value, we could specify that to the range function as the third argument.



```
# -*- coding: utf-8 -*-
"""
Created on Mon Jul 23 15:27:58 2018

@author: cd
"""

for i in range (1,20,5):
    print(i)
```

- Unfortunately, there's no such built-in function for floating point numbers. So, for example, there's no function that would allow us to create a list of the numbers from 0 to 0.72 with two consecutive numbers separated by 0.001. We can use a while loop to create our own function for this.



```
'''
Generate equally spaced floating point
numbers between two given values
'''
def frange(start, final, increment):
    numbers = []
    while start < final:
        numbers.append(start)
        start = start + increment
    return numbers
for i in frange (1,30,2.5):
              print(i)
```

Projectile Motion

- If we throw a ball across a field, it follows a trajectory like the one shown in the following Figure.

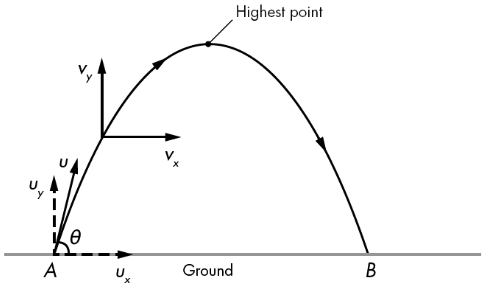


Figure 2-13: Motion of a ball that's thrown at point A*—at an angle (*θ*) with a velocity (*u*)—and that hits the ground at point* B

- In the figure, the ball is thrown from point A and lands at point B. This type of motion is referred to as projectile motion. Our aim here is to use the equations of projectile motion to graph the trajectory of a body, showing the position of the ball starting from the point it's thrown until it hits the ground again.
- Let's call the initial velocity u and the angle that it makes with the ground θ
- The ball has two velocity components: one along the x-direction, calculated by $u_x = ucos\theta$, and the other along the y-direction, where $u_y = usin\theta$.

- As the ball moves, its velocity changes, and we will represent that changed velocity using v: the horizontal component is v_x and the vertical component is v_y.
- For simplicity, assume the horizontal component (v_x) doesn't change during the motion of the body, whereas the vertical component (v_y) decreases because of the force of gravity according to the equation $v_y = u_y - gt$. In this equation, g is the gravitational acceleration and t is the time at which the velocity is measured.

- We have $u_y = usin\theta$, we can substitute to get $v_y = usin(\theta) - gt$.
- The horizontal component of the velocity remains constant, the horizontal distance traveled (S_x) is given by $S_x = u(cos\theta)t$
- The vertical component of the velocity changes, though, and the vertical distance traveled is given by the formula $S_y = usin(\theta)t - \frac{1}{2}gt^2$.
- S_x and S_y give us the $x-$ and $y-$coordinates of the ball at any given point in time during its flight. We'll use these equations when we write a program to draw the trajectory.

- 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 $t = \frac{usin\theta}{g}$

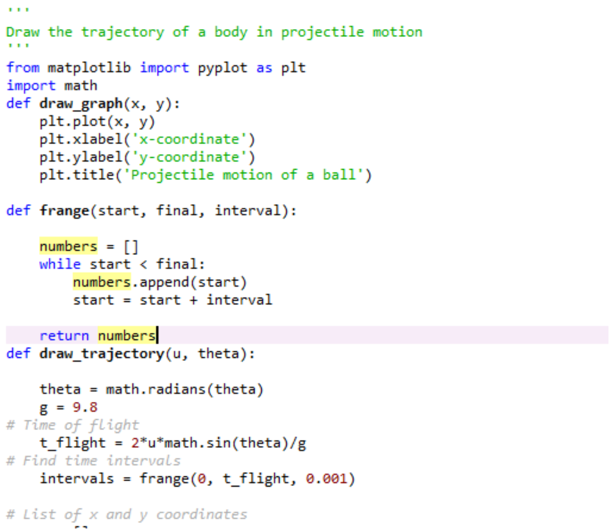
- We'll call this time t_{peak}. After it reaches its highest point, the ball will hit the ground after being airborne for another t_{peak} seconds, so the total time of flight (t_{flight}) of the ball is

$$t_{flight} = 2_{tpeak} = 2\frac{usin\theta}{g}$$

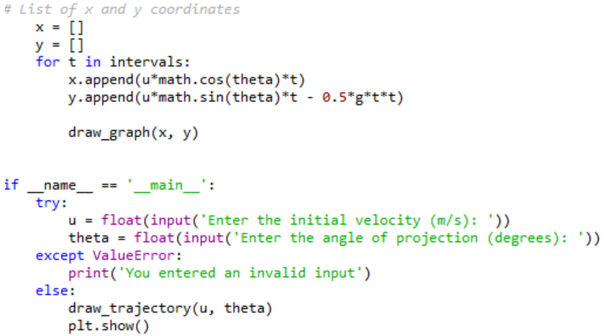
- Let's take a ball that's thrown with an initial velocity (u) of $5m/s$ at an angle (θ) of 45 degrees. To calculate the total time of flight, we substitute $u = 5, \theta = 45$, and $g = 9.8$ into the equation we saw above:

$$t_{flight} = 2\frac{5sin45}{9.8} = 0.72154sec$$

- 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.

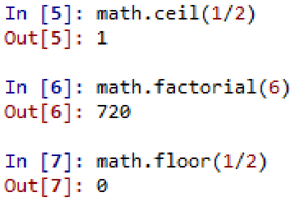


```
'''
Draw the trajectory of a body in projectile motion
'''
from matplotlib import pyplot as plt
import math
def draw_graph(x, y):
    plt.plot(x, y)
    plt.xlabel('x-coordinate')
    plt.ylabel('y-coordinate')
    plt.title('Projectile motion of a ball')

def frange(start, final, interval):

    numbers = []
    while start < final:
        numbers.append(start)
        start = start + interval

    return numbers
def draw_trajectory(u, theta):

    theta = math.radians(theta)
    g = 9.8
# Time of flight
    t_flight = 2*u*math.sin(theta)/g
# Find time intervals
    intervals = frange(0, t_flight, 0.001)

# List of x and y coordinates
```

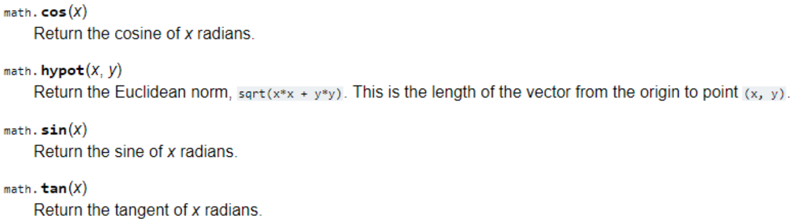


```python
# List of x and y coordinates
    x = []
    y = []
    for t in intervals:
        x.append(u*math.cos(theta)*t)
        y.append(u*math.sin(theta)*t - 0.5*g*t*t)

        draw_graph(x, y)


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

Math Module:

- math - Mathematical function
- This module is always available. It provides access to the mathematical functions defined by the *C* standard.
- These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers
- 

```
In [5]: math.ceil(1/2)
Out[5]: 1

In [6]: math.factorial(6)
Out[6]: 720

In [7]: math.floor(1/2)
Out[7]: 0
```

- 

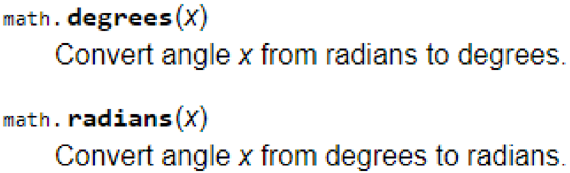
`math.cos(x)`
Return the cosine of *x* radians.

`math.hypot(x, y)`
Return the Euclidean norm, `sqrt(x*x + y*y)`. This is the length of the vector from the origin to point `(x, y)`.

`math.sin(x)`
Return the sine of *x* radians.

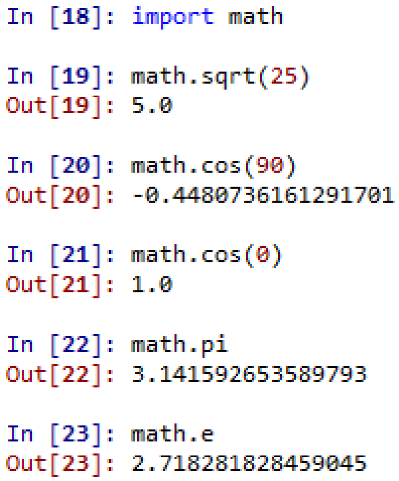
`math.tan(x)`
Return the tangent of *x* radians.

- Angular Conversion



`math.degrees(x)`
Convert angle *x* from radians to degrees.

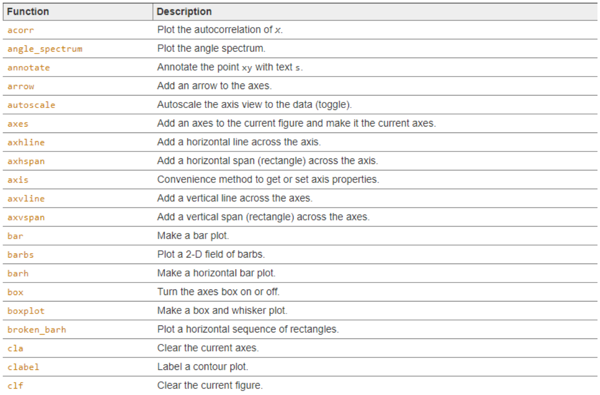
`math.radians(x)`
Convert angle *x* from degrees to radians.



```
In [18]: import math

In [19]: math.sqrt(25)
Out[19]: 5.0

In [20]: math.cos(90)
Out[20]: -0.4480736161291701

In [21]: math.cos(0)
Out[21]: 1.0

In [22]: math.pi
Out[22]: 3.141592653589793

In [23]: math.e
Out[23]: 2.718281828459045
```

The matplotlib.pyplot module

The matplotlib.pyplot module contains functions that allow you to generate many kinds of plots quickly

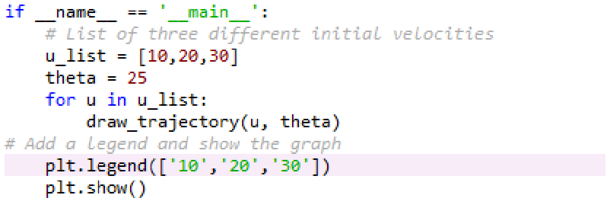
- 

Function	Description
`acorr`	Plot the autocorrelation of *x*.
`angle_spectrum`	Plot the angle spectrum.
`annotate`	Annotate the point `xy` with text `s`.
`arrow`	Add an arrow to the axes.
`autoscale`	Autoscale the axis view to the data (toggle).
`axes`	Add an axes to the current figure and make it the current axes.
`axhline`	Add a horizontal line across the axis.
`axhspan`	Add a horizontal span (rectangle) across the axis.
`axis`	Convenience method to get or set axis properties.
`axvline`	Add a vertical line across the axes.
`axvspan`	Add a vertical span (rectangle) across the axes.
`bar`	Make a bar plot.
`barbs`	Plot a 2-D field of barbs.
`barh`	Make a horizontal bar plot.
`box`	Turn the axes box on or off.
`boxplot`	Make a box and whisker plot.
`broken_barh`	Plot a horizontal sequence of rectangles.
`cla`	Clear the current axes.
`clabel`	Label a contour plot.
`clf`	Clear the current figure.

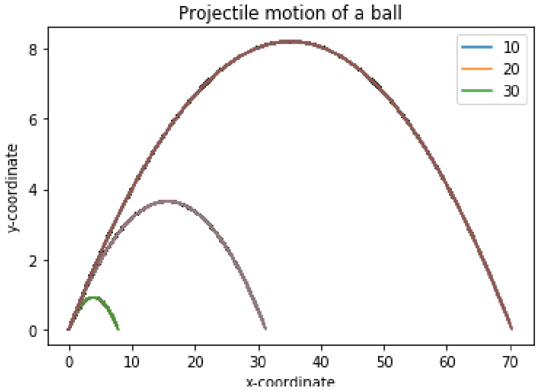
- https://matplotlib.org/api/pyplot_summary.html
- https://matplotlib.org/py-modindex.html

Comparing the Trajectory at Different Initial Velocities

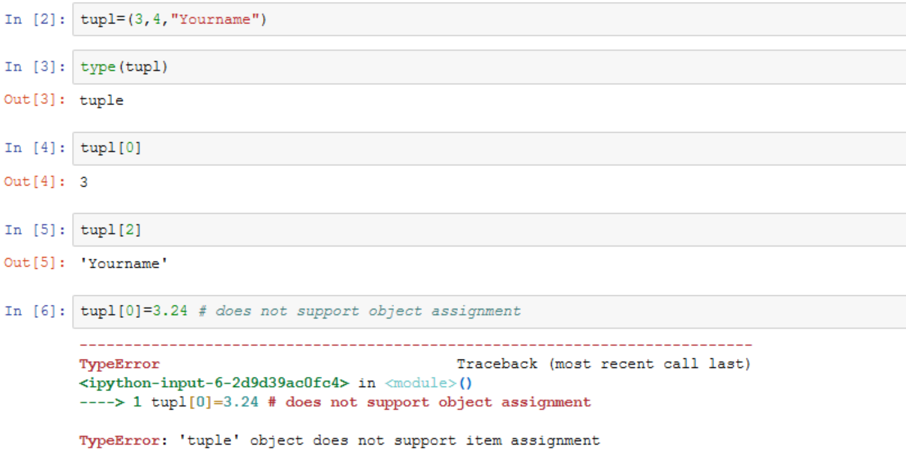
- The previous program allows us to perform interesting experiments. For example, what will the trajectory look like for three balls thrown at different velocities but with the same initial angle? To graph three trajectories at once, we can replace the main code block from our previous program with the following:



```python
if __name__ == '__main__':
    # List of three different initial velocities
    u_list = [10,20,30]
    theta = 25
    for u in u_list:
        draw_trajectory(u, theta)
# Add a legend and show the graph
    plt.legend(['10','20','30'])
    plt.show()
```

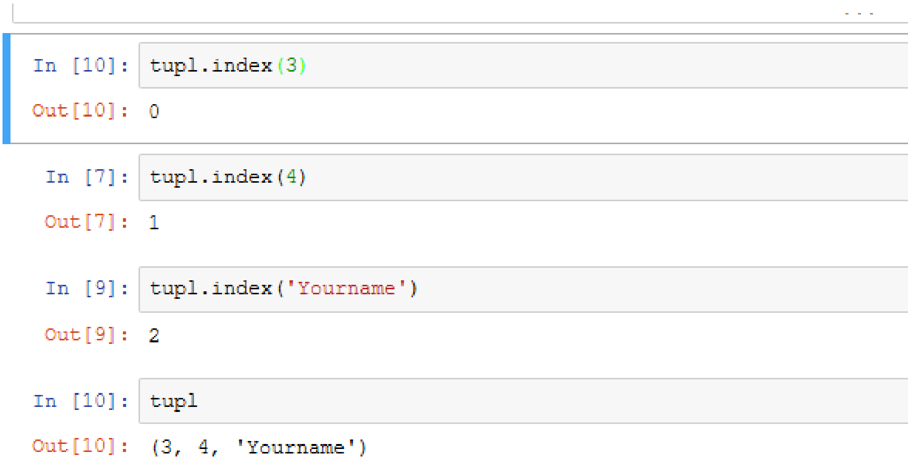

Projectile motion of a ball
10
20
30
y-coordinate
x-coordinate
0
2
4
6
8
0
10
20
30
40
50
60
70

List andTuples

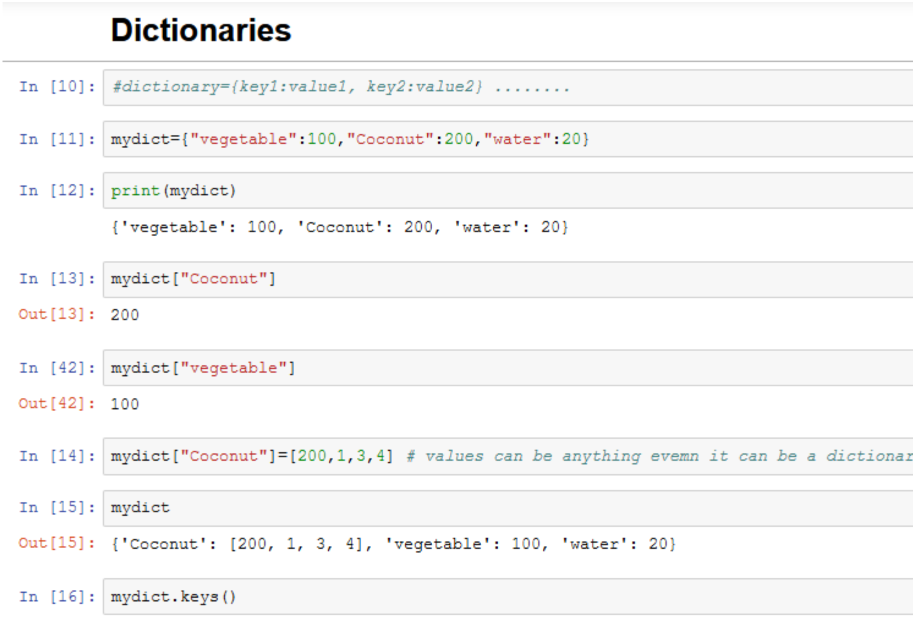


```
In [2]:  tupl=(3,4,"Yourname")

In [3]:  type(tupl)

Out[3]:  tuple

In [4]:  tupl[0]

Out[4]:  3

In [5]:  tupl[2]

Out[5]:  'Yourname'

In [6]:  tupl[0]=3.24 # does not support object assignment

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-6-2d9d39ac0fc4> in <module>()
         ----> 1 tupl[0]=3.24 # does not support object assignment

         TypeError: 'tuple' object does not support item assignment
```

- mutable -can change its value after using it. immutable cannot change its value after their creation
- Tuples are immutable
- Can not change the value after initialization.

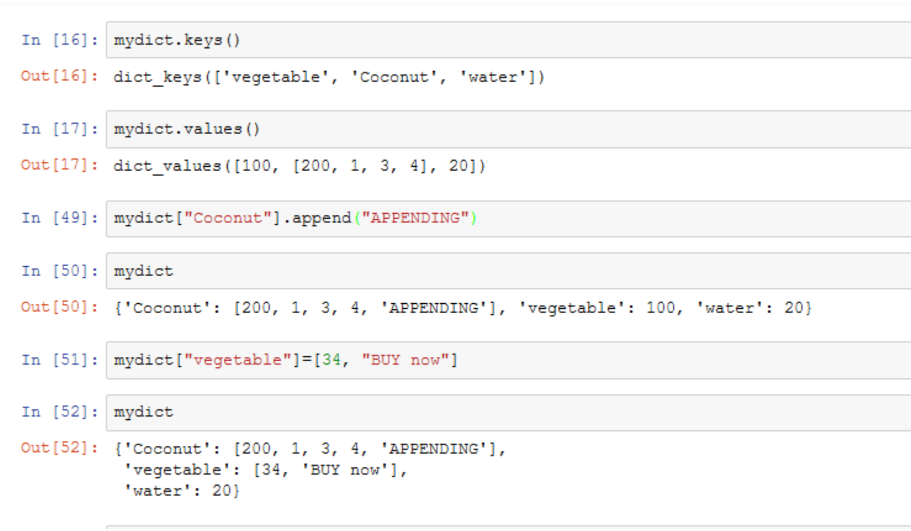


```
In [10]: tupl.index(3)
Out[10]: 0

In [7]: tupl.index(4)
Out[7]: 1

In [9]: tupl.index('Yourname')
Out[9]: 2

In [10]: tupl
Out[10]: (3, 4, 'Yourname')
```



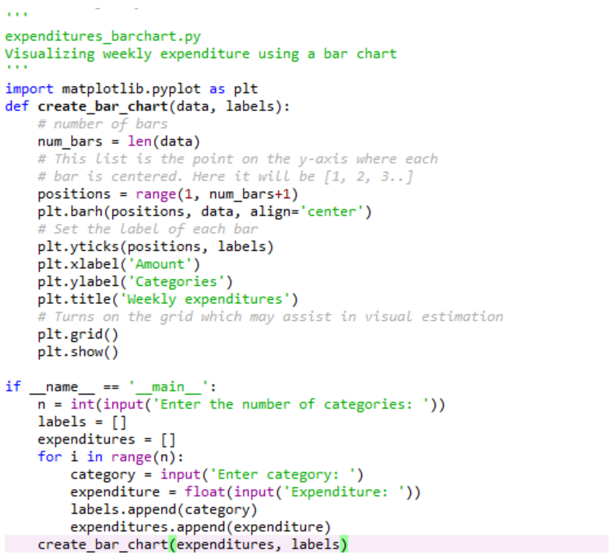
Dictionaries

```
In [10]: #dictionary={key1:value1, key2:value2} ........

In [11]: mydict={"vegetable":100,"Coconut":200,"water":20}

In [12]: print(mydict)

         {'vegetable': 100, 'Coconut': 200, 'water': 20}

In [13]: mydict["Coconut"]

Out[13]: 200

In [42]: mydict["vegetable"]

Out[42]: 100

In [14]: mydict["Coconut"]=[200,1,3,4] # values can be anything evemn it can be a dictionar

In [15]: mydict

Out[15]: {'Coconut': [200, 1, 3, 4], 'vegetable': 100, 'water': 20}

In [16]: mydict.keys()
```


```
In [16]: mydict.keys()

Out[16]: dict_keys(['vegetable', 'Coconut', 'water'])

In [17]: mydict.values()

Out[17]: dict_values([100, [200, 1, 3, 4], 20])

In [49]: mydict["Coconut"].append("APPENDING")

In [50]: mydict

Out[50]: {'Coconut': [200, 1, 3, 4, 'APPENDING'], 'vegetable': 100, 'water': 20}

In [51]: mydict["vegetable"]=[34, "BUY now"]

In [52]: mydict

Out[52]: {'Coconut': [200, 1, 3, 4, 'APPENDING'],
          'vegetable': [34, 'BUY now'],
          'water': 20}
```

Visualizing Your Expenses Using Bar Chart

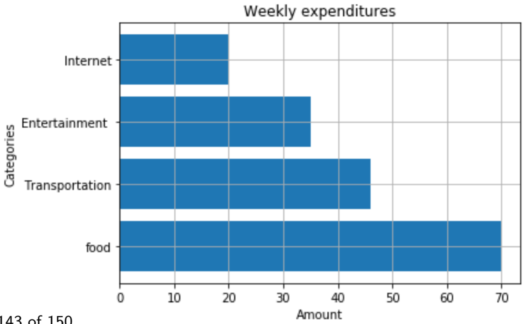
- The program should first ask for the number of categories for the expenditures and the weekly total expenditure in each category, and then it should create the bar chart showingthese expenditures. Here's a sample run of how the program should work:
- Enter the number of categories: 4
  Enter category: Food
  Expenditure: 70
  Enter category: Transportation
  Expenditure: 35
  Enter category: Entertainment
  Expenditure: 30
  Enter category: Phone/Internet
  Expenditure: 30



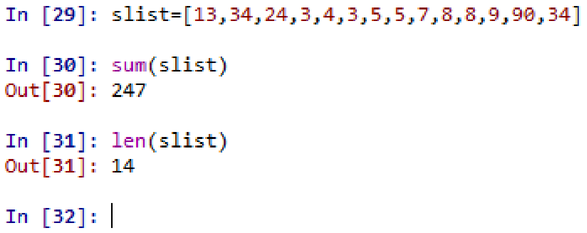
```python
'''
expenditures_barchart.py
Visualizing weekly expenditure using a bar chart
'''
import matplotlib.pyplot as plt
def create_bar_chart(data, labels):
    # number of bars
    num_bars = len(data)
    # This list is the point on the y-axis where each
    # bar is centered. Here it will be [1, 2, 3..]
    positions = range(1, num_bars+1)
    plt.barh(positions, data, align='center')
    # Set the label of each bar
    plt.yticks(positions, labels)
    plt.xlabel('Amount')
    plt.ylabel('Categories')
    plt.title('Weekly expenditures')
    # Turns on the grid which may assist in visual estimation
    plt.grid()
    plt.show()

if __name__ == '__main__':
    n = int(input('Enter the number of categories: '))
    labels = []
    expenditures = []
    for i in range(n):
        category = input('Enter category: ')
        expenditure = float(input('Expenditure: '))
        labels.append(category)
        expenditures.append(expenditure)
    create_bar_chart(expenditures, labels)
```

- The Fibonacci sequence $(1, 1, 2, 3, 5, ...)$ is the series of numbers where thei^{th} number in the series is the sum of the two previous numbers-that is, the numbers in the positions $(i-2)$ and $(i-1)$.

```
Enter the number of categories: 4

Enter category: food

Expenditure: 70

Enter category: Transportation

Expenditure: 46

Enter category: Entertainment

Expenditure: 35

Enter category: Internet

Expenditure: 20
```

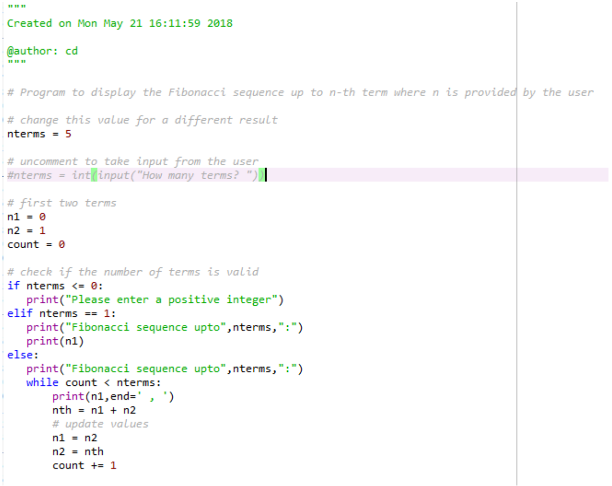


- When we use the *sum*() function on a list of numbers, it adds up all the numbers in the list and returns the result:
- We can use the len() function to give us the length of a list:

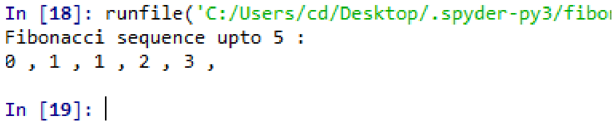


```
In [29]: slist=[13,34,24,3,4,3,5,5,7,8,8,9,90,34]

In [30]: sum(slist)
Out[30]: 247

In [31]: len(slist)
Out[31]: 14

In [32]: 
```

Program to display the Fibonacci sequence up to n-th term where n is provided by the user



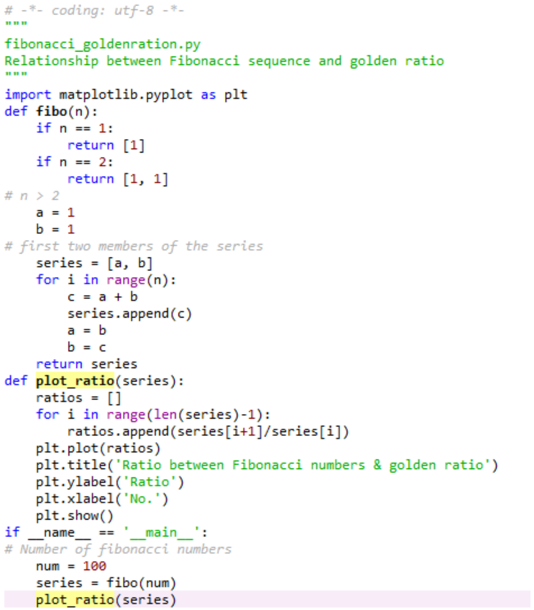
```
"""
Created on Mon May 21 16:11:59 2018

@author: cd
"""

# Program to display the Fibonacci sequence up to n-th term where n is provided by the user

# change this value for a different result
nterms = 5

# uncomment to take input from the user
#nterms = int(input("How many terms? "))

# first two terms
n1 = 0
n2 = 1
count = 0

# check if the number of terms is valid
if nterms <= 0:
   print("Please enter a positive integer")
elif nterms == 1:
   print("Fibonacci sequence upto",nterms,":")
   print(n1)
else:
   print("Fibonacci sequence upto",nterms,":")
   while count < nterms:
       print(n1,end=' , ')
       nth = n1 + n2
       # update values
       n1 = n2
       n2 = nth
       count += 1
```



```
In [18]: runfile('C:/Users/cd/Desktop/.spyder-py3/fibo
Fibonacci sequence upto 5 :
0 , 1 , 1 , 2 , 3 ,

In [19]:
```

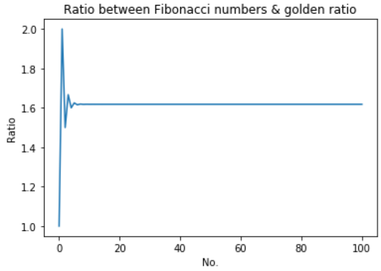
Exploring the Relationship Between the Fibonacci Sequence and the Golden Ratio

- In a Fibonacci sequence the ratios of consecutive pairs of numbers are nearly equal to each other. This value approaches a special number referred to as the *golden ratio= 1.618033988 . . .*.



```python
# -*- coding: utf-8 -*-
"""
fibonacci_goldenration.py
Relationship between Fibonacci sequence and golden ratio
"""
import matplotlib.pyplot as plt
def fibo(n):
    if n == 1:
        return [1]
    if n == 2:
        return [1, 1]
# n > 2
    a = 1
    b = 1
# first two members of the series
    series = [a, b]
    for i in range(n):
        c = a + b
        series.append(c)
        a = b
        b = c
    return series
def plot_ratio(series):
    ratios = []
    for i in range(len(series)-1):
        ratios.append(series[i+1]/series[i])
    plt.plot(ratios)
    plt.title('Ratio between Fibonacci numbers & golden ratio')
    plt.ylabel('Ratio')
    plt.xlabel('No.')
    plt.show()
if __name__ == '__main__':
# Number of fibonacci numbers
    num = 100
    series = fibo(num)
    plot_ratio(series)
```

```
In [21]: runfile('C:/Users/cd/Desktop/.spyder-py3/fibonaci.py', wdir='C:/Use
```



Thank you

THANK YOU