Learning Python

Hands-on Workshop @ Marwadi University, Rajkot

Instructor: Santosh Chapaneri

I. Python Concepts

Imports

In [1]:
```
# 'generic import' of math module
import math as mt
mt.sqrt(25)
```

Out[1]: 5.0

In [2]:
```
dir(mt)
```

Out[2]:
```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

In [3]:
```
# import a function
from math import sqrt
sqrt(25)    # no longer have to reference the module
```

Out[3]:
```
5.0
```

Data Types

In [4]:
```
# determine the type of an object
type(2)
```

Out[4]:
```
int
```

In [5]:
```
type(2.0)
```
Out[5]: float

In [6]:
```
type('two')
```
Out[6]: str

In [7]:
```
type(True)
```
Out[7]: bool

In [8]:
```
# convert an object to a given type
float(2)
```
Out[8]: 2.0

In [9]:
```
int(2.9)
```
Out[9]: 2

In [10]:
```
str(2.9)
```
Out[10]: '2.9'

Math in Python

In [11]:
```
10 + 4
```
Out[11]: 14

In [12]:
```
10 - 4
```
Out[12]: 6

In [13]:
```
10 * 4
```
Out[13]: 40

In [14]:
```
10 ** 4
```
Out[14]: 10000

In [15]:
```
5 % 4           # modulo
```
Out[15]: 1

In [16]:
```
10 / 4
```
Out[16]: 2.5

In [17]:
```
10 // 4 # floor division
```
Out[17]: 2

In [18]:
```
x = 2
print(x)
```
```
2
```

In [19]:
```
print(x * 4)
```
```
8
```

In [20]:
```
print(x**4)
```
```
16
```

Strings

properties: iterable, immutable

In [21]:
```
x = 4
print('This is a simple way of printing x', x)
```
```
This is a simple way of printing x 4
```

In [22]:
```
s = 'Python is fun!'
s[0]
```
Out[22]:
```
'P'
```

In [23]:
```
len(s)
```
Out[23]:
```
14
```

In [24]:
```
s.upper()
```
Out[24]:
```
'PYTHON IS FUN!'
```

In [25]:
```
s.find('fun')
```
Out[25]:
```
10
```

In [26]:
```
s.split()
```
Out[26]:
```
['Python', 'is', 'fun!']
```

Lists

properties: ordered, iterable, mutable, can contain multiple data types

In [27]:
```
x = list(range(10))
print(x)
x.reverse()
print(x)
x.append(5)
print(x)
x.sort()
print(x)
```
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 5]
[0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9]
```

In [28]:
```
x = list(range(5, 10))
x
```
Out[28]:
```
[5, 6, 7, 8, 9]
```

In [29]:
```
x = list(range(0, 10, 2))
x
```
Out[29]:
```
[0, 2, 4, 6, 8]
```

In [30]:
```
x = list(range(10, 0, -1))
x
```
Out[30]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Slicing Lists

[**start** : **stop** : **steps**]

[: stop] => slice from start till (excluding) stop index

[start :] => slice from start till end

-1 => means go backwards

In [31]:
```
numbers = [4, 7, 24, 11, 2]
print(numbers)
print(numbers[0:3])
print(numbers[-1])
print(numbers[-2])
print(numbers[3:])
print(numbers[:-1])
print(numbers[:4])
```

```
[4, 7, 24, 11, 2]
[4, 7, 24]
2
11
[11, 2]
[4, 7, 24, 11]
[4, 7, 24, 11]
```

EXERCISE:

1. Create a list of the first names of your family members. (>= 4 names)
2. Print the name of the last person in the list.
3. Print the length of the name of the first person in the list.
4. Change one of the names from their real name to their nickname.
5. Append a new person to the list.
6. Sort the list in reverse alphabetical order. (Use `sorted`)
7. Sort the list by the length of the names (shortest to longest). (Use `sorted`)

In [0]:
```
# 1. Create a list of the first names of your family members.
names = ['Ramesh', 'Suresh', 'Anil', 'Dhara', 'Sonali']
```

In [33]:
```
# 2. Print the name of the last person in the list.
names[-1]
```

Out[33]: `'Sonali'`

In [34]:
```
# 3. Print the length of the name of the first person in the list.
len(names[0])
```

Out[34]: `6`

In [35]:
```
# 4. Change one of the names from their real name to their nickname.
names[2] = 'Mac'
names[2]
```

Out[35]: `'Mac'`

In [36]:
```
# 5. Append a new person to the list.
names.append('Ruchi')
names
```

Out[36]: `['Ramesh', 'Suresh', 'Mac', 'Dhara', 'Sonali', 'Ruchi']`

In [37]:
```
# 6. Sort the list in reverse alphabetical order
sorted(names, reverse=True)
```

Out[37]: `['Suresh', 'Sonali', 'Ruchi', 'Ramesh', 'Mac', 'Dhara']`

In [38]:
```
# 7. Sort the list by the length of the names (shortest to longest)
sorted(names, key=len)
```

Out[38]: ['Mac', 'Dhara', 'Ruchi', 'Ramesh', 'Suresh', 'Sonali']

Dictionary

properties: unordered, iterable, mutable, can contain multiple data types

- made of key-value pairs
- keys must be unique, and can be strings, numbers, or tuples
- values can be any type

In [39]:
```
d = {'a': 1, 'b':2, 'c':3}
d
```

Out[39]: {'a': 1, 'b': 2, 'c': 3}

In [40]:
```
d['b']
```

Out[40]: 2

In [41]:
```
list(d.keys())
```

Out[41]: ['a', 'b', 'c']

In [42]:
```
list(d.values())
```

Out[42]: [1, 2, 3]

In [43]:
```
d['g'] = 7
d
```

Out[43]: {'a': 1, 'b': 2, 'c': 3, 'g': 7}

Dictionary Comprehension

In [44]:
```
a = { n: n*n for n in range(7) } # note curly brackets
print(a)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

List Comprehension

In [45]:
```
t = [x*3 for x in [5, 6, 7]]
print(t)
```

```
[15, 18, 21]
```

EXERCISE:

Q1: Given that: letters = ['a', 'b', 'c']. Write a list comprehension that returns: ['A', 'B', 'C']

Q2: Given that: word = 'abc' Write a list comprehension that returns: ['A', 'B', 'C']

Q3: Given that: fruits = ['Apple', 'Banana', 'Cherry'] Write a list comprehension that returns: ['A', 'B', 'C']

In [46]:
```
letters = ['a', 'b', 'c']
[letter.upper() for letter in letters]  # iterate through a list of strings,
                                        # and each string has an 'upper' method
```

Out[46]: ['A', 'B', 'C']

In [47]:
```
word = 'abc'
[letter.upper() for letter in word]     # iterate through each character
```
Out[47]:
```
['A', 'B', 'C']
```

In [48]:
```
fruits = ['Apple', 'Banana', 'Cherry']
[fruit[0] for fruit in fruits]
# slice the first character from each string
```
Out[48]:
```
['A', 'B', 'C']
```

Control Structures

For loops

In [49]:
```
x = [4, 3, 24, 7]
xsum = 0
for element in x:
    xsum += element # <--- this means xsum = xsum + element
print(xsum)
```
```
38
```

if-elif-else

In [50]:
```
x, y = 30, 20

if (x > y):
    print(x, '>', y)
elif (x == y):
    print(x, 'equals', y)
else:
    print('Hi there!')
```
```
30 > 20
```

While

In [51]:
```
i = 0
while (i < 5):
    print(i)
    i += 1
```
```
0
1
2
3
4
```

Functions

Python functions are defined using the **def** keyword.

In [0]:
```
def calc(a, b, op='add'):
    if op == 'add':
        return a + b
    elif op == 'sub':
        return a - b
    else:
        print('valid operations are add and sub')
```

In [53]:
```
calc(10, 4, op='add')
```
Out[53]: 14

In [54]:
```
calc(10, 4, 'add')  # unnamed arguments are inferred by position
```
Out[54]: 14

In [55]:
```
calc(10, 4)  # default 3rd param is used
```
Out[55]: 14

In [56]:
```
calc(10, 4, 'sub')
```
Out[56]: 6

In [57]:
```
calc(10, 4, 'div')
```
```
valid operations are add and sub
```

In [0]:
```
# return two values from a single function
def min_max(nums):
    return min(nums), max(nums)
```

In [59]:
```
nums = [1, 2, 3]
min_num, max_num = min_max(nums)
print(min_num)
print(max_num)
```
```
1
3
```

Classes

In [0]:
```
class Apollo:
    # Constructor
    def __init__(self, destination = "moon"):
        self.destination = destination

    # Methods
    def fly(self):
        print("Spaceship flying...")

    def get_destination(self):
        print("Destination is: " + self.destination)
```

Meaning of **"self"**

In [0]:
```
# 1st object
objFirst = Apollo()
# 2nd object
objSecond = Apollo()
```

In [0]:
```
# lets change the destination for objFirst to mars
objFirst.destination = "mars"
```

In [63]:
```
# objFirst calling fly function
objFirst.fly()
# objFirst calling get_destination function
objFirst.get_destination()
```
```
Spaceship flying...
Destination is: mars
```

In [64]:
```
# objSecond calling fly function
objSecond.fly()
# objSecond calling get_destination function
objSecond.get_destination()
```

```
Spaceship flying...
Destination is: moon
```

In [65]:
```
class BankAccount:
    """A simple bank account class"""

    def __init__(self, openingBalance):
        self.balance = openingBalance # account balance

    def deposit(self, amount): # makes deposit
        self.balance = self.balance + amount

    def withdraw(self, amount): # makes withdrawl
        self.balance = self.balance - amount

    def display(self): # displays balance
        print(self.balance)

ba1 = BankAccount(100.00) # create account

print('Before transactions:') # display balance
ba1.display()

ba1.deposit(74.35) # make deposit
ba1.withdraw(20.00) # make withdrawl

print('After transactions:') # display balance
ba1.display()
```

```
Before transactions:
100.0
After transactions:
154.35
```

II. NumPy

- Python lists are great. They can store strings, integers, or mixtures.
- NumPy arrays though are **multi-dimensional** and most **engineering** python libraries use them instead.
- They store the **same type of data** in each element and **cannot change size**.

In [66]:
```
import numpy as np

x = np.zeros(5)
print(x)
```

```
[0. 0. 0. 0. 0.]
```

In [67]:
```
x = np.zeros( (5,2) )
print(x)
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
```

In [68]:
```
print(np.arange(3, 10))       # Does not include end point
print(np.linspace(0, 1, 25))  # Includes end point
```

```
[3 4 5 6 7 8 9]
[0.         0.04166667 0.08333333 0.125      0.16666667 0.20833333
 0.25       0.29166667 0.33333333 0.375      0.41666667 0.45833333
 0.5        0.54166667 0.58333333 0.625      0.66666667 0.70833333
 0.75       0.79166667 0.83333333 0.875      0.91666667 0.95833333
 1.        ]
```

In [69]:
```
from math import pi
x = np.linspace(-pi, pi, 4)
print(np.cos(x))
```

```
[-1.   0.5  0.5 -1. ]
```

Numpy Broadcasting

- The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.

In [70]:
```
import numpy as np
a = np.array([[0.0,0.0,0.0], [10.0,10.0,10.0],
              [20.0,20.0,20.0], [30.0,30.0,30.0]])
b = np.array([0.0, 1.0, 2.0])

print('First array:')
print(a)
print('\n')

print('Second array:')
print(b)
print('\n')

print('First Array + Second Array')
print(a + b)
```

```
First array:
[[ 0.  0.  0.]
 [10. 10. 10.]
 [20. 20. 20.]
 [30. 30. 30.]]


Second array:
[0. 1. 2.]


First Array + Second Array
[[ 0.  1.  2.]
 [10. 11. 12.]
 [20. 21. 22.]
 [30. 31. 32.]]
```

EXERCISE:

Write functions to implement various activation methods used in machine learning: Sigmoid, ReLU, Softmax

Sigmoid

$$a_j = \sigma(x_j) = \frac{1}{1 + \exp(-x_j)}$$

ReLU (Rectified Linear Unit)

$$a_j = \sigma(x_j) = \max(0, x_j)$$

Softmax

$$a_j = \sigma(x_j) = \frac{\exp(x_j)}{\sum_k \exp(x_k)}$$

In [0]:
```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def relu(x):
    return np.maximum(0, x)

def softmax(x):
    return np.exp(x)/np.sum(np.exp(x))
```

In [72]:
```
x = np.array([1, 2, 3, 4, 1, 2, 3])
print(sigmoid(x))
print(relu(x))
print(softmax(x))
```

```
[0.73105858 0.88079708 0.95257413 0.98201379 0.73105858 0.88079708
 0.95257413]
[1 2 3 4 1 2 3]
[0.02364054 0.06426166 0.1746813  0.474833   0.02364054 0.06426166
 0.1746813 ]
```

- Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays.
- SciPy builds on this, and provides a large number of functions that operate on NumPy arrays and are useful for different types of **scientific and engineering applications**.

III. Scipy

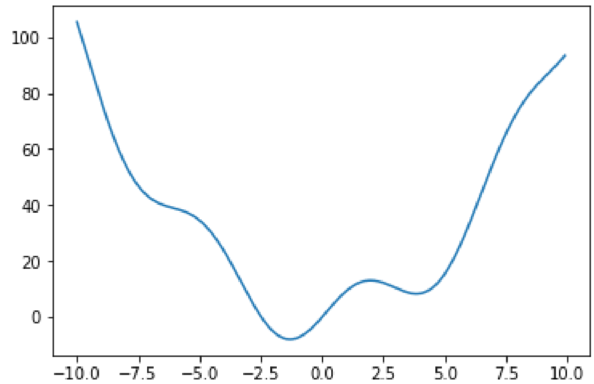
Optimization

In [73]:
```
# Example of Scipy functionality

# Optimization:
import scipy as sp
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**2 + 10*np.sin(x)

x = np.arange(-10, 10, 0.1)
plt.plot(x, f(x))
plt.show()
```



- This function has a global minimum around -1.3 and a local minimum around 3.8.
- Searching for minimum can be done with **scipy.optimize.minimize()**; given a starting point x_0, it returns the location of the minimum that it has found

In [74]:
```
from scipy.optimize import minimize
result = minimize(f, x0=0)
print(result)      # Global minimum
print(f(result.x)) # Value at global minimum
```

```
      fun: -7.945823375615215
 hess_inv: array([[0.08589237]])
      jac: array([-1.1920929e-06])
  message: 'Optimization terminated successfully.'
     nfev: 18
      nit: 5
     njev: 6
   status: 0
  success: True
        x: array([-1.30644012])
[-7.94582338]
```

IV. Pandas

In [0]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Dataframes

- According to Pandas documentation: *Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels.*
- In human terms, this means that a **dataframe has rows and columns**, can **change size**, and possibly **has mixed data types**.

Peek at the DataFrame contents

df.info() # index & data types

df.head(i) # get first i rows

df.tail(i) # get last i rows

df.describe() # summary stats cols

A very powerful feature in Pandas is **groupby**.

- This function allows us to **group together rows that have the same value in a particular column**.
- Then, we can aggregate this group-by object to compute statistics in each group.

MovieLens 100k movie rating data:

- main page: http://grouplens.org/datasets/movielens/ (http://grouplens.org/datasets/movielens/)
- 100,000 ratings from 1000 users on 1700 movies

In [76]:
```
from google.colab import files
uploaded = files.upload()
```

Choose Files No file chosen

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving u.user to u.user
```

In [0]:
```
import pandas as pd

import io
users = pd.read_csv(io.BytesIO(uploaded['u.user']), sep='|', index_col='user_id')
```

In [78]:
```
users.head()            # print the first 5 rows
```

Out[78]:

	age	gender	occupation	zip_code
user_id				
1	24	M	technician	85711
2	53	F	other	94043
3	23	M	writer	32067
4	24	M	technician	43537
5	33	F	other	15213

In [79]:
```
users.head(10)          # print the first 10 rows
```

Out[79]:

	age	gender	occupation	zip_code
user_id				
1	24	M	technician	85711
2	53	F	other	94043
3	23	M	writer	32067
4	24	M	technician	43537
5	33	F	other	15213
6	42	M	executive	98101
7	57	M	administrator	91344
8	36	M	administrator	05201
9	29	M	student	01002
10	53	M	lawyer	90703

In [80]:
```
users.tail()            # print the last 5 rows
```

Out[80]:

	age	gender	occupation	zip_code
user_id				
939	26	F	student	33319
940	32	M	administrator	02215
941	20	M	student	97229
942	48	F	librarian	78209
943	22	M	student	77841

In [81]:
```
users.index             # "the index" (aka "the labels")
```

Out[81]:
```
Int64Index([  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,
            ...
            934, 935, 936, 937, 938, 939, 940, 941, 942, 943],
           dtype='int64', name='user_id', length=943)
```

In [82]:
```
users.columns           # column names
```

Out[82]:
```
Index(['age', 'gender', 'occupation', 'zip_code'], dtype='object')
```

In [83]:
```
users.dtypes           # data types of each column
```

Out[83]:
```
age            int64
gender        object
occupation    object
zip_code      object
dtype: object
```

In [84]:
```
users.shape            # number of rows and columns
```

Out[84]:
```
(943, 4)
```

In [85]:
```
users.values           # underlying numpy array
```

Out[85]:
```
array([[24, 'M', 'technician', '85711'],
       [53, 'F', 'other', '94043'],
       [23, 'M', 'writer', '32067'],
       ...,
       [20, 'M', 'student', '97229'],
       [48, 'F', 'librarian', '78209'],
       [22, 'M', 'student', '77841']], dtype=object)
```

In [86]:
```
users.gender          # select one column using the DataFrame attribute
```

Out[86]:
```
user_id
1      M
2      F
3      M
4      M
5      F
6      M
7      M
8      M
9      M
10     M
11     F
12     F
13     M
14     M
15     F
16     M
17     M
18     F
19     M
20     F
21     M
22     M
23     F
24     F
25     M
26     M
27     F
28     M
29     M
30     M
      ..
914    F
915    M
916    M
917    F
918    M
919    M
920    F
921    F
922    F
923    M
924    M
925    F
926    M
927    M
928    M
929    M
930    F
931    M
932    M
933    M
934    M
935    M
936    M
937    M
938    F
939    F
940    M
941    M
942    F
943    M
Name: gender, Length: 943, dtype: object
```

In [87]:
```
# summarize (describe) the DataFrame
users.describe()                    # describe all numeric columns
```

Out[87]:

	age
count	943.000000
mean	34.051962
std	12.192740
min	7.000000
25%	25.000000
50%	31.000000
75%	43.000000
max	73.000000

In [88]:
```
users.describe(include=['object'])  # describe all object columns
```

Out[88]:

	gender	occupation	zip_code
count	943	943	943
unique	2	21	795
top	M	student	55414
freq	670	196	9

In [89]:
```
users.describe(include='all')       # describe all columns
```

Out[89]:

	age	gender	occupation	zip_code
count	943.000000	943	943	943
unique	NaN	2	21	795
top	NaN	M	student	55414
freq	NaN	670	196	9
mean	34.051962	NaN	NaN	NaN
std	12.192740	NaN	NaN	NaN
min	7.000000	NaN	NaN	NaN
25%	25.000000	NaN	NaN	NaN
50%	31.000000	NaN	NaN	NaN
75%	43.000000	NaN	NaN	NaN
max	73.000000	NaN	NaN	NaN

In [90]:
```
# count the number of occurrences of each value
users.gender.value_counts()     # most useful for categorical variables
```

Out[90]:
```
M    670
F    273
Name: gender, dtype: int64
```

In [91]:
```
users.age.value_counts()        # can also be used with numeric variables
```

Out[91]:
```
30    39
25    38
22    37
28    36
27    35
26    34
24    33
29    32
20    32
32    28
23    28
35    27
21    27
33    26
31    25
19    23
44    23
39    22
40    21
36    21
42    21
51    20
50    20
48    20
49    19
37    19
18    18
34    17
38    17
45    15
      ..
47    14
43    13
46    12
53    12
55    11
41    10
57     9
60     9
52     6
56     6
15     6
13     5
16     5
54     4
63     3
14     3
65     3
70     3
61     3
59     3
58     3
64     2
68     2
69     2
62     2
11     1
10     1
73     1
66     1
7      1
Name: age, Length: 61, dtype: int64
```

In [92]:
```
# Boolean filtering: only show users with age < 20
young_bool = users.age < 20         # create a Series of booleans...
users[young_bool]                   # ...and use that Series to filter rows
```

Out[92]:

	age	gender	occupation	zip_code
user_id				
30	7	M	student	55436
36	19	F	student	93117
52	18	F	student	55105
57	16	M	none	84010
67	17	M	student	60402
68	19	M	student	22904
101	15	M	student	05146
110	19	M	student	77840
142	13	M	other	48118
179	15	M	entertainment	20755
206	14	F	student	53115
221	19	M	student	20685
223	19	F	student	47906
246	19	M	student	28734
257	17	M	student	77005
258	19	F	student	77801
262	19	F	student	78264
270	18	F	student	63119
281	15	F	student	06059
289	11	M	none	94619
291	19	M	student	44106
303	19	M	student	14853
320	19	M	student	24060
341	17	F	student	44405
347	18	M	student	90210
367	17	M	student	37411
368	18	M	student	92113
375	17	M	entertainment	37777
393	19	M	student	83686
397	17	M	student	27514
...	...	...	...	...
601	19	F	artist	99687
609	13	F	student	55106
618	15	F	student	44212
619	17	M	student	44134
620	18	F	writer	81648
621	17	M	student	60402
624	19	M	student	30067
628	13	M	none	94306
631	18	F	student	38866
632	18	M	student	55454
642	18	F	student	95521

	age	gender	occupation	zip_code
user_id				
646	17	F	student	51250
674	13	F	student	55337
700	17	M	student	76309
710	19	M	student	92020
729	19	M	student	56567
747	19	M	other	93612
761	17	M	student	97302
787	18	F	student	98620
813	14	F	student	02136
817	19	M	student	60152
849	15	F	student	25652
851	18	M	other	29646
859	18	F	other	06492
863	17	M	student	60089
872	19	F	student	74078
880	13	M	student	83702
887	14	F	student	27249
904	17	F	student	61073
925	18	F	salesman	49036

77 rows × 4 columns

In [93]:
```
users[users.age < 20]              # or, combine into a single step
```

Out[93]:

	age	gender	occupation	zip_code
user_id				
30	7	M	student	55436
36	19	F	student	93117
52	18	F	student	55105
57	16	M	none	84010
67	17	M	student	60402
68	19	M	student	22904
101	15	M	student	05146
110	19	M	student	77840
142	13	M	other	48118
179	15	M	entertainment	20755
206	14	F	student	53115
221	19	M	student	20685
223	19	F	student	47906
246	19	M	student	28734
257	17	M	student	77005
258	19	F	student	77801
262	19	F	student	78264
270	18	F	student	63119
281	15	F	student	06059
289	11	M	none	94619
291	19	M	student	44106
303	19	M	student	14853
320	19	M	student	24060
341	17	F	student	44405
347	18	M	student	90210
367	17	M	student	37411
368	18	M	student	92113
375	17	M	entertainment	37777
393	19	M	student	83686
397	17	M	student	27514
...	...	...	...	...
601	19	F	artist	99687
609	13	F	student	55106
618	15	F	student	44212
619	17	M	student	44134
620	18	F	writer	81648
621	17	M	student	60402
624	19	M	student	30067
628	13	M	none	94306
631	18	F	student	38866
632	18	M	student	55454
642	18	F	student	95521

	age	gender	occupation	zip_code
user_id				
646	17	F	student	51250
674	13	F	student	55337
700	17	M	student	76309
710	19	M	student	92020
729	19	M	student	56567
747	19	M	other	93612
761	17	M	student	97302
787	18	F	student	98620
813	14	F	student	02136
817	19	M	student	60152
849	15	F	student	25652
851	18	M	other	29646
859	18	F	other	06492
863	17	M	student	60089
872	19	F	student	74078
880	13	M	student	83702
887	14	F	student	27249
904	17	F	student	61073
925	18	F	salesman	49036

77 rows × 4 columns

In [94]:
```
# for each occupation in 'users', count the number of occurrences
users.occupation.value_counts()
```

Out[94]:
```
student          196
other            105
educator          95
administrator     79
engineer          67
programmer        66
librarian         51
writer            45
executive         32
scientist         31
artist            28
technician        27
marketing         26
entertainment     18
healthcare        16
retired           14
salesman          12
lawyer            12
none               9
homemaker          7
doctor             7
Name: occupation, dtype: int64
```

In [95]:
```
# for each occupation, calculate the mean age
users.groupby('occupation').age.mean()
```

Out[95]:
```
occupation
administrator    38.746835
artist           31.392857
doctor           43.571429
educator         42.010526
engineer         36.388060
entertainment    29.222222
executive        38.718750
healthcare       41.562500
homemaker        32.571429
lawyer           36.750000
librarian        40.000000
marketing        37.615385
none             26.555556
other            34.523810
programmer       33.121212
retired          63.071429
salesman         35.666667
scientist        35.548387
student          22.081633
technician       33.148148
writer           36.311111
Name: age, dtype: float64
```

In [96]:
```
# for each occupation, calculate the minimum and maximum ages
users.groupby('occupation').age.agg(['min', 'max'])
```

Out[96]:

	min	max
occupation		
administrator	21	70
artist	19	48
doctor	28	64
educator	23	63
engineer	22	70
entertainment	15	50
executive	22	69
healthcare	22	62
homemaker	20	50
lawyer	21	53
librarian	23	69
marketing	24	55
none	11	55
other	13	64
programmer	20	63
retired	51	73
salesman	18	66
scientist	23	55
student	7	42
technician	21	55
writer	18	60

In [97]:
```
# for each combination of occupation and gender, calculate the mean age
users.groupby(['occupation', 'gender']).age.mean()
```

Out[97]:
```
occupation     gender
administrator  F         40.638889
               M         37.162791
artist         F         30.307692
               M         32.333333
doctor         M         43.571429
educator       F         39.115385
               M         43.101449
engineer       F         29.500000
               M         36.600000
entertainment  F         31.000000
               M         29.000000
executive      F         44.000000
               M         38.172414
healthcare     F         39.818182
               M         45.400000
homemaker      F         34.166667
               M         23.000000
lawyer         F         39.500000
               M         36.200000
librarian      F         40.000000
               M         40.000000
marketing      F         37.200000
               M         37.875000
none           F         36.500000
               M         18.600000
other          F         35.472222
               M         34.028986
programmer     F         32.166667
               M         33.216667
retired        F         70.000000
               M         62.538462
salesman       F         27.000000
               M         38.555556
scientist      F         28.333333
               M         36.321429
student        F         20.750000
               M         22.669118
technician     F         38.000000
               M         32.961538
writer         F         37.631579
               M         35.346154
Name: age, dtype: float64
```

Exercise: Pandas on IMDB Data

1. Read in 'imdb_1000.csv' and store it in a DataFrame named movies
2. Check the number of rows and columns
3. Check the data type of each column
4. Calculate the average movie duration
5. Sort by duration to find the shortest and longest movies (*Hint*: use `sort_values`)
6. Count how many movies have each of the content ratings
7. Calculate the average star rating for movies 2 hours or longer, and compare that with the average star rating for movies shorter than 2 hours
8. Calculate the average duration for each genre
9. Determine the top rated movie (by star rating) for each genre
10. Calculate the average star rating for each genre, but only include genres with at least 10 movies

In [98]:
```
from google.colab import files
uploaded = files.upload()
```

Choose Files No file chosen

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving imdb_1000.csv to imdb_1000.csv
```

In [0]:
```
import io
movies = pd.read_csv(io.BytesIO(uploaded['imdb_1000.csv']))
```

In [100]:
```
# 2. Check the number of rows and columns
movies.shape
```

Out[100]: (979, 6)

In [101]:
```
# 3. Check the data type of each column
movies.dtypes
```

Out[101]:
```
star_rating       float64
title              object
content_rating     object
genre              object
duration            int64
actors_list        object
dtype: object
```

In [102]:
```
# 4. Calculate the average movie duration
movies.duration.mean()
```

Out[102]: 120.97957099080695

In [103]:
```
# 5. Sort the DataFrame by duration to find the shortest and longest movies
movies.sort_values('duration').head(1)
```

Out[103]:

	star_rating	title	content_rating	genre	duration	actors_list
389	8.0	Freaks	UNRATED	Drama	64	[u'Wallace Ford', u'Leila Hyams', u'Olga Bacla...

In [104]:
```
movies.sort_values('duration').tail(1)
```

Out[104]:

	star_rating	title	content_rating	genre	duration	actors_list
476	7.8	Hamlet	PG-13	Drama	242	[u'Kenneth Branagh', u'Julie Christie', u'Dere...

In [105]:
```
# 6. Count how many movies have each of the content ratings
movies.content_rating.value_counts()
```

Out[105]:
```
R            460
PG-13        189
PG           123
NOT RATED     65
APPROVED      47
UNRATED       38
G             32
NC-17          7
PASSED         7
X              4
GP             3
TV-MA          1
Name: content_rating, dtype: int64
```

In [106]:
```
# 7. calculate the average star rating for movies 2 hours or longer,
# and compare that with the average star rating for movies shorter than 2 hours
movies[movies.duration >= 120].star_rating.mean()
```

Out[106]: 7.948898678414082

In [107]:
```
movies[movies.duration < 120].star_rating.mean()
```

Out[107]: 7.838666666666657

In [108]:
```
# 8. Calculate the average duration for each genre
movies.groupby('genre').duration.mean()
```

Out[108]:
```
genre
Action       126.485294
Adventure    134.840000
Animation     96.596774
Biography    131.844156
Comedy       107.602564
Crime        122.298387
Drama        126.539568
Family       107.500000
Fantasy      112.000000
Film-Noir     97.333333
History       66.000000
Horror       102.517241
Mystery      115.625000
Sci-Fi       109.000000
Thriller     114.200000
Western      136.666667
Name: duration, dtype: float64
```

In [109]:
```
# 9. Determine the top rated movie (by star rating) for each genre
movies.sort_values('star_rating', ascending=False).groupby('genre').title.first()
# movies.groupby('genre').title.first()
# equivalent, since DataFrame is already sorted by star rating
```

Out[109]:
```
genre
Action                                     The Dark Knight
Adventure    The Lord of the Rings: The Return of the King
Animation                                    Spirited Away
Biography                                 Schindler's List
Comedy                                        Modern Times
Crime                             The Shawshank Redemption
Drama                                         12 Angry Men
Family                          E.T. the Extra-Terrestrial
Fantasy                          The City of Lost Children
Film-Noir                                    The Third Man
History                                Battleship Potemkin
Horror                                              Psycho
Mystery                                        Rear Window
Sci-Fi                                        Blade Runner
Thriller                                 Shadow of a Doubt
Western                     The Good, the Bad and the Ugly
Name: title, dtype: object
```

In [110]:
```
# 10. Calculate the average star rating for each genre,
# but only include genres with at least 10 movies

# automatically create a list of relevant genres by saving the value_counts
# and then filtering
genre_counts = movies.genre.value_counts()
top_genres = genre_counts[genre_counts >= 10].index
movies[movies.genre.isin(top_genres)].groupby('genre').star_rating.mean()
```

Out[110]:
```
genre
Action       7.884559
Adventure    7.933333
Animation    7.914516
Biography    7.862338
Comedy       7.822436
Crime        7.916935
Drama        7.902518
Horror       7.806897
Mystery      7.975000
Name: star_rating, dtype: float64
```

V. Pickle

In [0]:
```
import pickle

# make an example object to pickle
some_obj = {'x':[4,2,1.5,1], 'y':[32,[101],17], 'foo':True, 'spam':False}
```

In [0]:
```
pickle.dump( some_obj, open( "mypickle.p", "wb" ) )
```

In [0]:
```
del some_obj
# Delete from memory
```

In [114]:
```
loaded_obj = pickle.load( open( "mypickle.p", "rb" ) )
loaded_obj
```

Out[114]:
```
{'foo': True, 'spam': False, 'x': [4, 2, 1.5, 1], 'y': [32, [101], 17]}
```

VI. Errors and Exceptions

Programming errors:

```
**Syntax errors**: Errors where the code is not valid Python (generally easy to fix)
**Runtime errors**: Errors where syntactically valid code fails to execute, perhaps due to invalid user
input (sometimes easy to fix)
**Semantic errors**: Errors in logic: code executes without a problem, but the result is not what you e
xpect (often very difficult to track-down and fix)
```

Let's focus on **Runtime Errors**

In [115]:
```
# referencing an undefined variable
print(Q)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-115-cbf1bd89097d> in <module>()
----> 1 print(Q)

NameError: name 'Q' is not defined
```

In [116]:
```
# trying an operation that's not defined
1 + 'abc'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-116-a51a3635a212> in <module>()
----> 1 1 + 'abc'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [117]:
```
# compute a mathematically ill-defined result
2/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-117-e8326a161779> in <module>()
----> 1 2/0

ZeroDivisionError: division by zero
```

In [118]:
```
# access a sequence element that doesn't exist
L = [1, 2, 3]
L[1000]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-118-354067ebdc84> in <module>()
      1 L = [1, 2, 3]
----> 2 L[1000]

IndexError: list index out of range
```

in each case, Python is kind enough to not simply indicate that an error happened, but to spit out a meaningful exception that includes information about what exactly went wrong, along with the exact line of code where the error happened.

Having access to meaningful errors like this is immensely useful when trying to trace the root of problems in your code.

Catching Exceptions: *try* and *except*

In [119]:
```
try:
    print("let's try something:")
    x = 1 / 0 # ZeroDivisionError
except:
    print("something bad happened!")
```

```
let's try something:
something bad happened!
```

In [0]:
```
def safe_divide(a, b):
    try:
        return a / b
    except:
        return 1E100
```

In [121]:
```
safe_divide(1, 2)
```

Out[121]: 0.5

In [122]:
```
safe_divide(2, 0)
```

Out[122]: 1e+100

Raising Exceptions: *raise*

In [123]:
```
raise RuntimeError("my error message")
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-123-b1834d213d3b> in <module>()
----> 1 raise RuntimeError("my error message")

RuntimeError: my error message
```

In [0]:
```
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

One potential problem in above Fibonacci function is that the input value could be negative.

This will not currently cause any error in our function, but we might want to let the user know that a negative N is not supported.

Errors stemming from invalid parameter values, by convention, lead to a ValueError being raised:

In [0]:
```
def fibonacci(N):
    if N < 0:
        raise ValueError("N must be non-negative")
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

In [126]:
```
fibonacci(10)
```

Out[126]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

In [127]:
```
fibonacci(-10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-127-f1ae0a8066f0> in <module>()
----> 1 fibonacci(-10)

<ipython-input-125-721cefef37c2> in fibonacci(N)
      1 def fibonacci(N):
      2     if N < 0:
----> 3         raise ValueError("N must be non-negative")
      4     L = []
      5     a, b = 0, 1

ValueError: N must be non-negative
```

In [128]:
```
N = -10
try:
    print("trying this...")
    print(fibonacci(N))
except ValueError:
    print("Bad value: need to do something else")
```

```
trying this...
Bad value: need to do something else
```