Python

October 7, 2018

1 1 Introduction

Python is high level programming created by "Guido Van Rossum" and released in 1991. Main philosophy of the python is to increase the readability of the code. Python supports Object Oriented Programming, Imperative Functional and Procedural programming paradigms. Python is managed by non-profit organization so it is developed under OSI approved open source license.

Python is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs.

The language's core philosophy is: * Beautiful is better than ugly * Explicit is better than implicit * Simple is better than complex * Complex is better than complicated * Readability counts

Rather than having all of its functionality built into its core, Python was designed to be highly extensible. This compact modularity has made it particularly popular as a means of adding programmable interfaces to existing applications. Van Rossum's vision of a small core language with a large standard library and easily extensible interpreter stemmed from his frustrations with ABC, which espoused the opposite approach.

While offering choice in coding methodology, the Python philosophy rejects exuberant syntax (such as that of Perl) in favor of a simpler, less-cluttered grammar. As Alex Martelli put it: "To describe something as 'clever' is not considered a compliment in the Python culture." Python's philosophy rejects the Perl "there is more than one way to do it" approach to language design in favor of "there should be one—and preferably only one—obvious way to do it"

2 2 Data Types

Following are the data types available in python that help us to solve our problems. ## 2.1 Variables * Variable is a label of memory location. * In python variables don't need explicit declaration to occupy memory space. * The type of the variable will be decided after assigning the value to it. * In Python, we may reuse the same variable to store values of any type

Why

```
To store data
```

When

```
When you have or need data you can store it ina variable
```

Where

`Veriables can be declaired every where`

Some examples are given below

```
In [17]: counter = 100.00 #floating type variable
         miles = 100 #integer type variable
         name = "Mehvish" #string type variable

         #print all variables
         print(counter)
         print(miles)
         print(name)
```

```
100.0
100
Mehvish
```

```
In [13]: name = "Jin Kazama" #string type variable
         age = 24 #integer type variable
         sex = "Male" #string type variables
         height = 5.5 #floating type variable

         #print all variables
         print(name)
         print(age)
         print(sex)
         print(height)
```

```
Jin Kazama
24
Male
5.5
```

```
In [15]: tracking_id = "tracking 1" #string type variable
         trackable_id = 552 #integer type variable
         location = "30.3753,69.3451" #string type variable

         #print all variables
         print("Tracking Id : " + tracking_id)
         print("Trackable Id : " + str(trackable_id))
         print("Location : " + location)
```

```
Tracking Id : tracking 1
Trackable Id : 552
Location : 30.3753,69.3451
```

```
In [16]: book_id = 1 #integer type variable
         book_author = "Khalil Ul Rehman" #string type variable
         book_publisher = "Pak Pub" #string type variable

         #print all variables
         print(book_id)
         print(book_author)
         print(book_publisher)
```

```
1
Khalil Ul Rehman
Pak Pub
```

2.1 2.2 Numbers

- This data type stores numeric values.
- They are immutable data types (Means changing the value of Number data type will result in newly allocated object).
- There are four different types of number or numeric type which are following
  - Plain Integers
    - They also called integers and have at least 32 bits precision.
    - Integers are implemented using long in C.
    - Booleans are also subtype of plain integers.
  - Long Integers
    - Long integers have unlimited precision.
  - Floating Point Numbers
    - Floating numbers are implemented using double in C.
  - Complex Numbers
    - Complex numbers have a real and imaginary part, which are each implemented using double in C.

Why

```
To store numbers data type
```

When

```
When you need or want to store some data for future accesment you can use this variable
```

Where

```
In any part of the code you can create a variable
```

Some Examples of usages of numbers are given below.

```
In [18]: 2 #integer
         print(2)
```

```
2
```

```
In [19]: 2+3 #integer
         print(2+3)
```

```
5
```

```
In [22]: 100-86 #integer
         print(100-86)
```

```
14
```

```
In [ ]: 58*40 #integer
        print(58*40)
```

```
In [24]: 49//2 #integer ((floored) quotient of 49 and 2)
         print(49//2)
```

```
24
```

```
In [25]: 59%2 #integer (remainder of 59 / 2)
         print(59%2)
```

```
1
```

```
In [27]: abs(895) #integer (absolute value or magnitude of 895)
         print(abs(895))
```

```
895
```

```
In [23]: 36/6 #floating number
         print(36/6)
```

```
6.0
```

```
In [26]: abs(12.89) #floating number (absolute value or magnitude of 12.89)
         print(abs(12.89))
```

```
12.89
```

```
In [29]: complex(5,99) #complex number (a complex number with real part 5, imaginary part 99)
         print(complex(5,99))
         print(complex(5,99).conjugate()) # (conjugate of complex number)
```

```
(5+99j)
(5-99j)
```

```
In [32]: divmod(69,5) #the pair (69 // 5, 69 % 5)
         print(divmod(69,5))
```

```
(13, 4)
```

```
In [34]: pow(2,10) # 2 pow 10 function
         print (pow(2,10))

         2**10 #exponent (same as power)
         print(2**10)
```

```
1024
1024
```

```
In [35]: 2.5 + 5.5 #floating point number
         print(2.5+5.5)
```

```
8.0
```

2.2 2.3 String

- String data types are used to store words or combination of words having letters, numbers, special characters etc.
- Python doesn't support char data type. It will be as String of length one in Python.
- String literals are written in a variety of ways.
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes".
  - Triple quoted: '''Three single quotes''', """Three double quotes"""
    - Triple quoted string can span multiple lines it will include whitespaces in it.

Why

```
To store string data
```

When

```
When you need to store analyse or want to use string data later your can use.
```

Where

```
Use can create string variable in any part of the code.
```

Some examples of string is given below.

```
In [36]: name = "Mehvish" #string
         university = "COMSATS" #string

         print(name,university)
```

```
Mehvish COMSATS
```

```
In [37]: location = "31.3753,68.3451"
         name = "Location On Earth"
         print(location,name)
```

```
31.3753,68.3451 Location On Earth
```

```
In [38]: subject = "Machine Learning"
         gpa = "4.0"
         student_name = "Khalil Ul Rehman"

         print(student_name,gpa,subject)
```

```
Khalil Ul Rehman 4.0 Machine Learning
```

```
In [39]: expression = "5+8 = 999007760"
         print(expression)
```

```
5+8 = 999007760
```

2.2.1 2.3.1 Access Values in String

- Python doesn't support Character data type. These are treated as String of length one.
- Square brackets are used to access substrings.
- We can access a specific character or range of characters from string.

It would be cleared by using following examples.

```
In [41]: stringVariable = 'Hello Word'

         print("stringVariable[0]:",stringVariable[0])
         print("stringVariable[1:5]",stringVariable[1:5])
```

```
stringVariable[0]: H
stringVariable[1:5] ello
```

```
In [1]: stringVariable = "This is Watch"

        print("stringVariable[5]" , stringVariable[5])
```

```
stringVariable[5] i
```

```
In [4]: stringVariable = "its python tutorial"
        print(len(stringVariable))
```

```
19
```

```
In [6]: stringVariable = ['A','B','C','D','E','F','G','H']
        print(len(stringVariable))
        print(stringVariable[3:len(stringVariable)])
```

```
8
['D', 'E', 'F', 'G', 'H']
```

2.2.2 2.3.2 Updating String

Value of the string can be updated by assigning a new value to it. New value will be replaced with older.

New value must be related (having same data type) to previous value of the string. '+' Operator is used for the concatenation of string. Examples for string Updating given below.

```
In [8]: stringVariable = "Hello Word"
        stringVariable = "Python"
        print(stringVariable)
```

```
Python
```

```
In [7]: stringVariable = "Hello Word"
        stringVariable = stringVariable[:6] + "Python"
        print(stringVariable)
```

```
Hello Python
```

```
In [3]: stringVariable = "Hello Word"
        stringVariable = ""
        print(stringVariable)
```

```
In [4]: stringVariable = "Here I am"
        stringVariable = stringVariable[7] + "k"
        print(stringVariable)
```

```
ak
```

2.2.3 2.3.3 Delete String

To delete the value of the string just delete its object (Reference Variable).
Examples are given below.

```
In [6]: stringVariable = "Hello Python"
        print(stringVariable)
        del stringVariable
        print(stringVariable)

Hello Python



        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-6-4f5a8fea46a0> in <module>()
          2 print(stringVariable)
          3 del stringVariable
    ----> 4 print(stringVariable)


        NameError: name 'stringVariable' is not defined


In [7]: name = "Khalil Ul Rehman"
        print(name)
        del name
        print(name)

Khalil Ul Rehman



        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-7-d8785f7da6f4> in <module>()
          2 print(name)
          3 del name
    ----> 4 print(name)


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

```
In [8]: testingString = 'Here is another string'
        print(testingString)
        del testingString
        print(testingString)
```

```
Here is another string
```

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

        <ipython-input-8-bf10899bf9fc> in <module>()
          2 print(testingString)
          3 del testingString
    ----> 4 print(testingString)


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

2.2.4 2.3.4 String Special Operators

Some special operators are used in Python to assist the user. These are given below.

- '+' Operator is used for concatenation. Example is given below.

```
In [11]: variable = "Hello"
         print(variable + "Python")
```

```
HelloPython
```

```
In [12]: name = "Hameed"
         name = name+ " Ahmad"
         print(name)
```

```
Hameed Ahmad
```

```
In [10]: stringVariable = "Khalil"
         print(stringVariable + ' is a software engineer')
```

```
Khalil is a software engineer
```

```
In [14]: stringVariable = "I can code in"
         stringVariable += " Python"
         print(stringVariable)
```

```
I can code in Python
```

- '*' Operator is used to repeat string. Examples are given below

```
In [15]: variable = "Hello"
         print(variable*3)
```

```
HelloHelloHello
```

```
In [19]: stringVariable = 'I'
         print(stringVariable + (' Love'*10) + " Pakistan")
```

```
I Love Love Love Love Love Love Love Love Love Love Pakistan
```

```
In [20]: dot = "."
         print("Loading" + dot*3)
```

```
Loading...
```

```
In [22]: variable = "You need to wait"
         print(variable + '.'*50)
```

```
You need to wait...
```

- '[]' Give the character of string at given index. Examples are given below.

```
In [23]: variable = "Hello"
         print(variable[1])
```

```
e
```

```
In [25]: stringVariable = ["q","e","g"]
         print(stringVariable[0])
```

```
q
```

```
In [27]: stringVariable = "Here we go"
         print(stringVariable[3])
```

```
e
```

```
In [28]: stringVariable = "Khalil"
         print(stringVariable[5])
```

```
l
```

- '[:]' is use to get a range of characters from string. Examples are given below.

```
In [29]: variable = "Hello"
         print(variable[1:3])
```

```
el
```

```
In [31]: stringVariable = "I am Khalil"
         print(stringVariable[4:11])
```

```
 Khalil
```

```
In [33]: variable = "Include"
         print(variable[0:1])
```

```
I
```

```
In [34]: stringVariable = "Exclude"
         print(stringVariable[3:5])
```

```
lu
```

- 'in' returns true if given character exists in string, otherwise false. Examples are given below.

```
In [35]: variable = "Hello"
         print( "H" in variable)
```

```
True
```

```
In [36]: stringVariable = "I am Khalil"
         print("Khalil" in stringVariable)
```

```
True
```

```
In [40]: stringVariavle = "I am khalil"
         print("Khail" in stringVariable)
```

```
False
```

```
In [45]: stringVariable = "I aM KhAliL"
         print("khalil" in stringVariable.lower())
```

```
True
```

- 'not in' returns true if given character does not exists in string, Otherwise false. Examples are given below.

```
In [46]: variable = "Hello"
         print( "H" not in variable)
```

```
False
```

```
In [47]: stringVariable = "I am Khalil"
         print("Khalil" not in stringVariable)
```

```
False
```

```
In [48]: stringVariavle = "I am khalil"
         print("Khail" not in stringVariable)
```

```
True
```

```
In [50]: stringVariable = "I aM KhAliL"
         print("khalil" not in stringVariable.lower())
```

```
False
```

2.2.5 2.3.5 String Formatting Operator

One of the coolest feature is string formatting operator within print statement. Examples are given below.

```
In [51]: print("My name is %s and age is %d years" % ("Zara" , 26))
```

```
My name is Zara and age is 26 years
```

```
In [52]: print("We are taking about %s. The year of production is %s" % ("Milk Pack","2010"))
```

```
We are taking about Milk Pack. The year of production is 2010
```

```
In [53]: print("I am living in %s. It cost me RS%d" % ("Lahore",30000))
```

```
I am living in Lahore. It cost me RS30000
```

```
In [55]: print("This rod is made of %s and its heigh is approximately %f'" % ("Wood",5.5))
```

```
This rod is made of Wood and its heigh is approximately 5.500000'
```

2.3 2.4 Lists

- It can be written as a list of comma-separated values within square brackets.
- Multiple types of data can be stored in a List.
- Individual element can be change (It can be read and write).
- List indices start from 0 just like arrays.
- Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).
- Only pair of brackets are used to denote empty list.

Why

```
To sotre similar type of data
```

When

```
When you need to store large amount of data of same datatype
```

Where

```
You can declare any where in side the code blocks
```

```
In [56]: list1 = ["Mehvish","1880"]
         print(list1)
```

```
['Mehvish', '1880']
```

```
In [57]: list1 = ["Bucket","2''5'","RS 3500"]
         print(list1)
```

```
['Bucket', "2''5'", 'RS 3500']
```

```
In [58]: anotherList = [500,400,852,684,7,7]
         print(anotherList)
```

```
[500, 400, 852, 684, 7, 7]
```

```
In [59]: floatingPointList = [5.5,155.5,1.1,78.39]
         print(floatingPointList)
```

```
[5.5, 155.5, 1.1, 78.39]
```

2.3.1 2.4.1 Accessing Values in Lists

- Square brackets are used to access the values of a list.
- We can access a specific values or range of values from list.

```
In [66]: list1 = ["Physics","Chemistry",1997,2000]
         list2 = [1,2,3,4,5,6,7]

         print("list1[0]",list1[0])
         print("list2[1:5]",list2[1:5])
```

```
list1[0] Physics
list2[1:5] [2, 3, 4, 5]
```

```
In [68]: human = ["Khalil","Muhammad Mushtaq",24,5.5,"Software Engineer"]
         print("Name : %s \nFather Name : %s\nAge : %d\nHeight : %f\nEducation : %s" % (human[
```

```
Name : Khalil
Father Name : Muhammad Mushtaq
Age : 24
Height : 5.500000
Education : Software Engineer
```

```
In [70]: product = ["Bucket","Office",1200,"optp"]
         print("Production Company is",product[3])
```

```
Production Company is optp
```

```
In [71]: car = ["R1",24000,12000000,"Sports Car"]
         print("Category :",car[3])
```

```
Category : Sports Car
```

2.3.2 2.4.2 Updating Lists

- We can update single or multiple elements of a list by giving slice on left hand of the assignment operator.
- It can also be updated using append() function.

```
In [72]: list1 = ["Physics","Chemistry",1997,2000]
         list1[1] = "Computer Science"

         print(list1)

         list1.append("Computer Science")
         print(list1)
```

```
['Physics', 'Computer Science', 1997, 2000]
['Physics', 'Computer Science', 1997, 2000, 'Computer Science']
```

```
In [73]: universityList = ["COMSATS","NUST","FAST"]
         universityList.append("UET")
         print(universityList)
```

```
['COMSATS', 'NUST', 'FAST', 'UET']
```

```
In [74]: companiesList = ["Next Bridge","Net Sol","Systems Engineer"]
         companiesList[2] = "Systems Limited"
         print(companiesList)
```

```
['Next Bridge', 'Net Sol', 'Systems Limited']
```

```
In [75]: listABC = [1,2,3,4,5,6,7,8,9,25,89,63]
         listABC[3] = 63
         listABC.append(500)
         print(listABC)
```

```
[1, 2, 3, 63, 5, 6, 7, 8, 9, 25, 89, 63, 500]
```

2.3.3 2.4.3 Delete List Element

- del statement is used to remove when you know the position or index of element to be deleted.
- remove() function can be used when you don't know the position of element to be removed.

```
In [76]: list1 = ["Physics","Chemistry",1997,2000]
         print(list1)

         del list1[2];

         print("list after deleting value at index 2")
         print(list1)
```

```
['Physics', 'Chemistry', 1997, 2000]
list after deleting value at index 2
['Physics', 'Chemistry', 2000]
```

```
In [77]: list1 = ["Physics","Chemistry",1997,2000]
         print(list1)

         del list1[1];

         print("list after deleting value at index 1")
         print(list1)
```

```
['Physics', 'Chemistry', 1997, 2000]
list after deleting value at index 1
['Physics', 1997, 2000]
```

```
In [78]: list1 = ["Physics","Chemistry",1997,2000]
         print(list1)

         del list1[3];

         print("list after deleting value at index 3")
         print(list1)
```

```
['Physics', 'Chemistry', 1997, 2000]
list after deleting value at index 3
['Physics', 'Chemistry', 1997]
```

```
In [79]: list1 = ["Physics","Chemistry",1997,2000]
         print(list1)

         del list1[0];

         print("list after deleting value at index 0")
         print(list1)
```

```
['Physics', 'Chemistry', 1997, 2000]
list after deleting value at index 0
['Chemistry', 1997, 2000]
```

```
In [80]: list1 = ["Physics","Chemistry",1997,2000]
         print(list1)

         list1.remove("Chemistry");

         print("list after deleting value")
         print(list1)
```

```
['Physics', 'Chemistry', 1997, 2000]
list after deleting value
['Physics', 1997, 2000]
```

```
In [81]: list1 = ["Physics","Chemistry",1997,2000]
         print(list1)

         list1.remove("Physics");

         print("list after deleting value")
         print(list1)
```

```
['Physics', 'Chemistry', 1997, 2000]
list after deleting value
['Chemistry', 1997, 2000]
```

```
In [82]: list1 = ["Physics","Chemistry",1997,2000]
         print(list1)

         list1.remove(1997);

         print("list after deleting value")
         print(list1)
```

```
['Physics', 'Chemistry', 1997, 2000]
list after deleting value
['Physics', 'Chemistry', 2000]
```

```
In [83]: list1 = ["Physics","Chemistry",1997,2000]
         print(list1)

         list1.remove(2000);

         print("list after deleting value")
         print(list1)
```

```
['Physics', 'Chemistry', 1997, 2000]
list after deleting value
['Physics', 'Chemistry', 1997]
```

2.4 2.5 Dictionaries

- It is save as array of objects having a key and values in PHP.
- Each Key is separated by colon (:) from its value.
- Each item is separated with comma (,).
- Empty dictionary can be written as {}.
- Keys are unique in dictionary but values may not be.
- The values of dictionary can be of any data type.
- The key must be of an immutable data type such as strings, numbers, or tuples.

Why

```
To store data with given index/key.
```

When

```
We use it when we have key-value pairs and we need to store that.
```

Where

```
Same as lists we can can create it in any code block and it will be accesseble in that code blo
```

2.4.1 2.5.1 Accessing Value in Dictionary

We can use the familiar square brackets along with the key to obtain its value.
Examples are given below.

```
In [1]: dict = {"name":"Mehvish","depart":"BCS","batch":2014}
        print(dict['name'])
```

```
Mehvish
```

```
In [3]: product = { "id":1,"name":"ram","des":"random access memory","price":15000 }
        print(" ID",product["id"],"\n","Name",product["name"],"\n","Price",product["price"])
```

```
 ID 1
 Name ram
 Price 15000
```

```
In [4]: tracking = { "id":"1a","trackable-id":1,"start-time":"22:05","meet-time":"03:50"}
        print(tracking["id"],tracking["start-time"])
```

```
1a 22:05
```

```
In [5]: dict = {"name":"lotsti","des":"Its somthing special"}
        print(dict["des"])
```

```
Its somthing special
```

2.4.2 2.5.2 Updating Dictionary

- We can update already existing value in dictionary.
- We can also add a new entry in dictionary.

Examples are below

```
In [6]: dict = {"name":"Mehvish","depart":"BCS","batch":2014}
        print(dict)
        dict["depart"] = "Computer Science"
        print(dict)
        dict["name"] = "Mehvish Hayat"
        print(dict)
```

```
{'name': 'Mehvish', 'depart': 'BCS', 'batch': 2014}
{'name': 'Mehvish', 'depart': 'Computer Science', 'batch': 2014}
{'name': 'Mehvish Hayat', 'depart': 'Computer Science', 'batch': 2014}
```

```
In [7]: product = { "id":1,"name":"ram","des":"random access memory","price":15000 }
        print(" ID",product["id"],"\n","Name",product["name"],"\n","Price",product["price"])
        product["des"] = "Random Selection Memory"
        print(product)
        product["price"] = 50500
        print(product)
```

```
 ID 1
 Name ram
 Price 15000
{'id': 1, 'name': 'ram', 'des': 'Random Selection Memory', 'price': 15000}
{'id': 1, 'name': 'ram', 'des': 'Random Selection Memory', 'price': 50500}
```

```
In [8]: tracking = { "id":"1a","trackable-id":1,"start-time":"22:05","meet-time":"03:50"}
        print(tracking["id"],tracking["start-time"])
        tracking["meet-time"] = "01:10"
        print(tracking)
```

```
1a 22:05
{'id': '1a', 'trackable-id': 1, 'start-time': '22:05', 'meet-time': '01:10'}
```

```
In [9]: dict = {"name":"lotsti","des":"Its somthing special"}
        print(dict["des"])
        dict["des"] = "Its good"
        print(dict)
```

```
Its somthing special
{'name': 'lotsti', 'des': 'Its good'}
```

2.4.3 2.5.3 Delete Dictionary Element

- We can delete individual element of dictionary and complete content of dictionary.
- del is used for individual element removal and clear() function is used to remove entire dictionary.

```
In [11]: dict = {"name":"Mehvish","depart":"BCS","batch":2014}
         print(dict)

         del dict["batch"]
         print(dict)

         dict.clear()
         print(dict)
```

```
{'name': 'Mehvish', 'depart': 'BCS', 'batch': 2014}
{'name': 'Mehvish', 'depart': 'BCS'}
{}
```

```
In [12]: product = { "id":1,"name":"ram","des":"random access memory","price":15000 }
         print(" ID",product["id"],"\n","Name",product["name"],"\n","Price",product["price"])

         del product["name"]
         print(product)

         product.clear()
         print(product)

 ID 1
 Name ram
 Price 15000
{'id': 1, 'des': 'random access memory', 'price': 15000}
{}
```

```
In [13]: tracking = { "id":"1a","trackable-id":1,"start-time":"22:05","meet-time":"03:50"}
         print(tracking["id"],tracking["start-time"])

         del tracking["meet-time"]
         print(tracking)

         tracking.clear()
         print(tracking)

1a 22:05
{'id': '1a', 'trackable-id': 1, 'start-time': '22:05'}
{}
```

```
In [14]: dict = {"name":"lotsti","des":"Its somthing special"}
         print(dict["des"])

         del dict["des"]
         print(dict)

         dict.clear()
         print(dict)

Its somthing special
{'name': 'lotsti'}
{}
```

2.5 2.6 Tuples

- It is same as list. The difference between tuples and list are, the tuples cannot be changed.
- Each item is comma (,) separated.
- Empty Tuple is shown as ().

- To write a tuple containing a single value you have to include a comma (,) even though there is only one value. For example: tip = (40,).
- It can also have multiple data type values.
- Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Why

```
To store same type of data.
```

When

```
We use it when we need to store same type of data.
```

Where

```
Same as lists we can can create it in any code block and it will be accesseble in that code blo
```

2.5.1 2.6.1 Accessing Values in Tuples

We can use the familiar square brackets along with the index to obtain its value. Following are simple example for understanding.

```
In [15]: tup = ("Math","98","C Programming","99")
         print(tup)

         print(tup[1])
         print(tup[1:3])
```

```
('Math', '98', 'C Programming', '99')
98
('98', 'C Programming')
```

```
In [20]: subjects = ("Fundamental Programming","Object Oriented Pogramming","Design Pattern","S
         print(subjects)
         print(subjects[2:10])
```

```
('Fundamental Programming', 'Object Oriented Pogramming', 'Design Pattern', 'Software Construct
('Design Pattern', 'Software Construction')
```

```
In [21]: companies = ("samsung","htc","motorola","apple")
         print(companies)
```

```
('samsung', 'htc', 'motorola', 'apple')
```

```
In [22]: series = (2,4,6,8,10)
         print(series)
```

```
(2, 4, 6, 8, 10)
```

2.5.2 2.6.2 Updating Tuples

- Tuples are immutable means we can't change it. It is read-only.
- We are able to take portions of tuples to make a new tuple.

Examples are given below in which we will see how is it possible to change tuples.

```
In [23]: tup1 = (12,34.56)
         tup2 = ('abc','xyz')

         #Following Action is Not Valid for Tuples
         #tup1[1] = 1244

         #so lets create new tuple
         tup3 = tup1 + tup2
         print(tup3)
```

```
(12, 34.56, 'abc', 'xyz')
```

```
In [25]: subjects = ("Fundamental Programming","Object Oriented Pogramming","Design Pattern","S
         print(subjects)
         subjects = subjects + ("Software Testing",)
         print (subjects)
```

```
('Fundamental Programming', 'Object Oriented Pogramming', 'Design Pattern', 'Software Construct
('Fundamental Programming', 'Object Oriented Pogramming', 'Design Pattern', 'Software Construct
```

```
In [27]: companies = ("samsung","htc","motorola","apple")
         print(companies)
         companies = companies[1:2] + ("New Company",)
         print(companies)
```

```
('samsung', 'htc', 'motorola', 'apple')
('htc', 'New Company')
```

```
In [28]: series = (2,4,6,8,10)
         print(series)
         series = ("def",) + ("loki","thor")
         print(series)
```

```
(2, 4, 6, 8, 10)
('def', 'loki', 'thor')
```

2.5.3 2.6.3 Delete Tuple Elements

- Removing individual element in Tuple is not possible because they can't be updated.
- del statement is used to remove entire Tuple.

Examples are given below.

```
In [30]: tup = ("Physics","Chemistry",1997,2000)
         print(tup)

         del tup
         print("After Deleting Tup")
         print(tup)
```

```
('Physics', 'Chemistry', 1997, 2000)
After Deleting Tup
```

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

        NameError                                 Traceback (most recent call last)

        <ipython-input-30-253f73e8b706> in <module>()
          4 del tup
          5 print("After Deleting Tup")
    ----> 6 print(tup)


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

```
In [34]: subjects = ("Fundamental Programming","Object Oriented Pogramming","Design Pattern","S
         print(subjects)
         del subjeccts
         print(subjects)
```

```
('Fundamental Programming', 'Object Oriented Pogramming', 'Design Pattern', 'Software Construct
```

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

        NameError                                 Traceback (most recent call last)

        <ipython-input-34-314528f4d5fd> in <module>()
          1 subjects = ("Fundamental Programming","Object Oriented Pogramming","Design Pattern"
          2 print(subjects)
    ----> 3 del subjeccts
```

```
  4 print(subjects)


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

```
In [35]: companies = ("samsung","htc","motorola","apple")
         print(companies)
         del companies
         print(companies)

('samsung', 'htc', 'motorola', 'apple')
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-35-d28fe23bc907> in <module>()
  2 print(companies)
  3 del companies
----> 4 print(companies)


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

```
In [36]: series = (2,4,6,8,10)
         print(series)
         del series
         print(series)

(2, 4, 6, 8, 10)
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-36-e1aa97707c0a> in <module>()
  2 print(series)
  3 del series
----> 4 print(series)


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

2.6 2.7 Sets

- A set is collection of unordered items.
- Every element is unique (No duplication).
- Every element is immutable (can't be changed).
- However, the set itself is mutable We can add or remove items from it.
- Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.
- Empty set can be written as {}.
- Each item in set will be comma separated.
- We can make a set from a list using set() function.
- Data type can be found using type() function.
- add() is used to add single value, upgrade() is used for adding multiple values.
- update() function can take tuple, strings, list or other set as argument. In all cases, duplications will be avoided.
- discard() and remove() functions are used to delete particular item from set.
- discard() will not raise an error if item doesn't exists in set.
- remove() will raise an error if item doesn't exits in set.

Why

```
To store different type of data in large amount that is collectively of one category.
```

When

```
We use it when we have different type of data in large amount.
```

Where

```
Same as lists we can can create it in any code block and it will be accesseble in that code bl
```

Examples for understanding are given below.

```
In [1]: #list
        list1 = [1,2,3,4.5]

        print( type(list1) )

        my_set = set(list1)
        print(my_set)

        print(type(my_set))

        #set of integer
        my_set = {1,2,3}
        print(my_set)

        #set of mixed data types
        my_set = {1,"Hello",1.2,"C"}
```

```
        #adding one velue
        my_set.add("D")
        print(my_set)
        #adding multiple values
        my_set.update(list1)
        print(my_set)

        my_set.discard("G")
        print(my_set)

        my_set.remove("G")
```

```
<class 'list'>
{1, 2, 3, 4.5}
<class 'set'>
{1, 2, 3}
{1, 'C', 1.2, 'D', 'Hello'}
{1, 2, 'C', 3, 4.5, 1.2, 'D', 'Hello'}
{1, 2, 'C', 3, 4.5, 1.2, 'D', 'Hello'}
```

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

        KeyError                                  Traceback (most recent call last)

        <ipython-input-1-8cba7977b5da> in <module>()
         26 print(my_set)
         27
    ---> 28 my_set.remove("G")


        KeyError: 'G'
```

```
In [47]: integer_set = {1,45,10}
         print(integer_set)

         integer_set.add(9)
         print(integer_set)
```

```
{1, 10, 45}
{1, 10, 45, 9}
```

```
In [1]: set_of_shapes = {"Square","Cube","Circle"}
        print(set_of_shapes)
```

```
        set_of_shapes.update({"Triangle","Spher","Cylendar"})
        print(set_of_shapes)
```

```
{'Square', 'Circle', 'Cube'}
{'Triangle', 'Cylendar', 'Spher', 'Square', 'Circle', 'Cube'}
```

```
In [2]: set_of_direction = {"Up","Down"}
        print(set_of_direction)

        set_of_direction.add("Forward")
        set_of_direction.add("Backword")

        print(set_of_direction)

        set_of_direction.update({"Left","Right"})
        print(set_of_direction)
```

```
{'Up', 'Down'}
{'Forward', 'Backword', 'Up', 'Down'}
{'Backword', 'Down', 'Forward', 'Up', 'Right', 'Left'}
```

3 3 Comparison Operators

These are used to compare values (string or number) and return true/false according to situation.

Why

```
To compare values that is some value is equal or greater or lesser then other
```

When

```
We use it when we want to make decission
```

Where

```
Most of the time we use these operations in decissions statements
```

Examples are given below.

```
In [3]: 1<3
```

```
Out[3]: True
```

```
In [4]: 15<=233
```

```
Out[4]: True
```

```
In [5]: "Mehvish" == "Zeenat"
```

```
Out[5]: False

In [6]: "Mehvish" != "Mehvish"

Out[6]: False

In [7]: (1==1) or (5>2)

Out[7]: True

In [8]: (1<2) and (2<1)

Out[8]: False

In [9]: 1==1 or 2>1 and 1<2

Out[9]: True

In [10]: "Khail" != "Usman" and 500==500

Out[10]: True

In [11]: "khalil" == "KhalIl"

Out[11]: False
```

4 4 If-Else Statements

If-Else statements are used to execute a block of code depending on conditions. 'if' block will execute when 'if' statement will be true otherwise 'else' block will execute. It is explained in below examples.

Why

```
To make decission where a code block will be executed or not.
```

When

```
We use it when we need to decide about the next execuation.
```

Where

```
We can can create it in any code block.
```

```
In [15]: if 25 % 2:
             print("even")
         else:
             print("odd")

even
```

```
In [17]: age = 17
         if age < 18:
             print("Teen")
         elif age < 23:
             print("Young")
         elif age < 50:
             print("Adult")
         else:
             print("old")
```

```
Teen
```

```
In [18]: if "Khalil" == "khalil":
             print("Same Name")
         else:
             print("Not Same Name")
```

```
Not Same Name
```

```
In [19]: if "Khalil" == "Khalil":
             print("Same Name")
         else:
             print("Not Same Name")
```

```
Same Name
```

5 5 For and While Loop

Python has for and while loop for iteration, used when we want to perform a specific a task repeatedly.

Why

```
To repeate particular code.
```

When

```
We use it when we need to repeate specific code for multiple time.
```

Where

```
We can can create it in any code block.
```

```
In [ ]: # For loop Examples
```

```
In [24]: fact = 1
         N = 5
         for i in range(1,N+1):
             fact*=i
         print(fact)
```

```
120
```

```
In [25]: for i in range (1,10):
             print("*")
```

```
*
*
*
*
*
*
*
*
*
```

```
In [28]: for i in range (1,10):
             print("*"*i)
```

```
*
**
***
****
*****
******
*******
********
*********
```

```
In [29]: for i in range (1,10):
             print("*"* (10-i))
```

```
*********
********
*******
******
*****
****
***
**
*
```

```
In [ ]: # While Loop Examples
```

```
In [30]: a = 0
         while a < 10:
             a = a+1
             print(a)
```

```
1
2
3
4
5
6
7
8
9
10
```

```
In [33]: a = 10
         while a > 0:
             a -= 2
             print(a)
```

```
8
6
4
2
0
```

```
In [35]: trigger = "action"

         while trigger == "action":
             print("Inside loop")
             trigger = "out of action"
```

```
Inside loop
```

```
In [36]: xyz = 500
         while xyz > 0:
             xyz -= 125
             print(xyz)
```

```
375
250
125
0
```

6 6 Functions

- It is a block of organized and reusable code.
- It is used to perform a single, related action.

- It provides high modularity for your application.
- It has a high degree of code reusing.

The syntax is:
Def functionName(parameters) : function_docstring
Function_suite Return [expression]

Why

To reuse code block.

When

We use it when we have some code that we need to use in different places.

Where

We write defination in class or in side normal code block

```
In [37]: # function defination is here
         def printme( str ):
             #this will print the string that will passed to this function
             print (str)
             return
         #now you can call user premitive function
         printme("I'm first call to user defined function!")
```

```
I'm first call to user defined function!
```

```
In [38]: def iseven(number):
             return number % 2 == 0;
         print ("56 is even",iseven(56))
```

```
56 is even True
```

```
In [40]: def iscontain(str1,str2):
             return str2 in str1

         print (iscontain("Khalil","lil"))
```

```
True
```

```
In [42]: def sum(value1 , value2):
             return value1 + value2
         print ("Sum of 12 and 56",sum(12,56))
```

```
Sum of 12 and 56 68
```

7 7 Lambda Functions

The creation of anonymous functions at runtime, using a construct called “lambda”.

Lambda function doesn’t include return statement, it always contains an expression which is returned.

This piece of code shows the difference between a normal function definition (“f”) and a lambda function (“g”).

```
In [45]: #Normal function
         def f (x):
             return x**2;
         print (f(8))

         #Lambda function
         #Lambda Epression
         times3 = lambda var:var*3
         times3(10)
         #Lambda expression = another way to write function in one line
```

```
64
```

```
Out[45]: 30
```

```
In [46]: iseven = lambda var:var%2 == 0
         print(iseven(56))
```

```
True
```

```
In [47]: convertLOwerCase = lambda var:var.lower()
         print(convertLOwerCase("KhaLIL"))
```

```
khalil
```

```
In [48]: area = lambda radious:2 * 3.14 * radious
         print("Area of Circle ",area(2))
```

```
Area of Circle  12.56
```

7.1 7.1 Map()

- Map() function is used with two arguments. Just like: r = map(func, seq)
- The first argument func is the name of a function and the second a sequence (e.g. a list).
- Seq. map() applies the function func to all the elements of the sequence seq. It returns a new list with the elements changed by func.

```
In [49]: sentence = "It is raning cats and dogs"
         words = sentence.split()
         print(words)

         length = map( lambda word: len(word) ,words )
         list (length)
```

```
['It', 'is', 'raning', 'cats', 'and', 'dogs']
```

```
Out[49]: [2, 2, 6, 4, 3, 4]
```

```
In [51]: human = ["Khalil",24,"Male",5.6]
         types = map( lambda var:type(var) , human)
         list(types)
```

```
Out[51]: [str, int, str, float]
```

```
In [53]: sequence = [1,2,3,4,5,6,7,8]
         evens = map( lambda var:var%2==0  , sequence)
         list(evens)
```

```
Out[53]: [False, True, False, True, False, True, False, True]
```

```
In [54]: names = ["kHalil","AAmer","mEhvish","zeNat"]
         name_lower = map( lambda name:name.lower() , names)
         list(name_lower)
```

```
Out[54]: ['khalil', 'aamer', 'mehvish', 'zenat']
```

7.2 7.2 Filter()

- The function filter(function, list) offers an elegant way to filter out all the elements of a list.
- The function filter(f,l) needs a function f as its first argument. F returns a Boolean value, i.e. either True or False.
- This function will be applied t every element of the list l.
- Only if f returns True will the element of the list be included in the result list.

```
In [56]: fib = [0,1,1,2,3,5,8,13,21,34,55]
         result1 = filter( lambda x: x%2 , fib)
         list(result1)
```

```
Out[56]: [1, 1, 3, 5, 13, 21, 55]
```

```
In [57]: fib = [0,1,1,2,3,5,8,13,21,34,55]
         result1 = filter( lambda x: x%2 == 0 , fib)
         list(result1)
```

```
Out[57]: [0, 2, 8, 34]
```

```
In [58]: names = ["khalil","zulker","zenat","kali"]
         filtered_name = filter( lambda x: "k" in x , names)
         list(filtered_name)

Out[58]: ['khalil', 'zulker', 'kali']

In [60]: values = [500,556,25,6,2,63,555,36,36,3,56,6,63,6563,66,256,56]
         filtered_values = filter( lambda v: v < 100 , values)
         list(filtered_values)

Out[60]: [25, 6, 2, 63, 36, 36, 3, 56, 6, 63, 66, 56]

In [61]: values = [500,556,25,6,2,63,555,36,36,3,56,6,63,6563,66,256,56]
         filtered_values = filter( lambda v: v > 100 , values)
         list(filtered_values)

Out[61]: [500, 556, 555, 6563, 256]
```

8 8 File I/O

In this section, we'll cover all basic I/O function (methods).

Why

```
To read or write data in files.
```

When

```
We use it when we need to read or write data in files.
```

Where

```
We can can create it in any code block.
```

8.1 8.1 Reading input from keyboard

- For reading input from keyboard, raw_input() method is used.
- It reads only one line from standard input and returns it as a string.

```
In [63]: from six.moves import input
         string  = input("Enter Your Name : ")
         print(string)

Enter Your Name : Mehvish
Mehvish


In [64]: from six.moves import input
         name = input("Enter your name: ")
         age = input ("Enter your age: ")

         print( name, age)
```

```
Enter your name: Khalil
Enter your age: 24
Khalil 24
```

```
In [65]: from six.moves import input
         string = input("Enter your country")
         print(string)
```

```
Enter your countryPakistan
Pakistan
```

```
In [68]: from six.moves import input
         number = input("Enter number : ")
         number = int(number)
         if number%2 == 0:
             print(number,"is even")
         else:
             print(number,"is not even")
```

```
Enter number : 53
53 is not even
```

8.2 8.2 I/O from or to Text File

In this scenario, we'll read and write to a text file. * r opens a file in read only mode. * r+ opens a file read and write mode. * w opens a file in write mode only. * a opens a file in append mode. * a+ opens a file in append and read mode.

```
In [71]: #open a file to read
         fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","r+")
         str = fileOpen.read() #to read specific content from file. read(12) will return 12 ch
         print(str)
         fileOpen.close()
```

```
Name: Mehvish Ashiq
Department: BSCS
```

```
In [72]: #open a file to read
         fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","r+")
         str = fileOpen.read(5) #to read specific content from file. read(12) will return 12 c
         print(str)
         fileOpen.close()
```

```
Name:
```

```
In [74]: #open a file to read
         fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","r+")
         str = fileOpen.read(10) #to read specific content from file. read(12) will return 12
         print(str)
         fileOpen.close()
```

```
Name: Mehv
```

```
In [3]: #open a file to read
        fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","r+")
        str = fileOpen.read(8) #to read specific content from file. read(12) will return 12 ch
        print(str)
        fileOpen.close()
```

```
Name: Me
```

```
In [6]: #Open a file to append
        fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","a+")
        fileOpen.write(" Information Technology Lahore")
        fileOpen.close()

        #open a file to read
        fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","r+")
        string = fileOpen.read() #to read specific content from start you can use read(12). It
        print(string)
        #close opend file
        fileOpen.close()
```

```
Name: Mehvish Ashiq
Department: BSCS
 Computer Science
 Computer Science Information Technology Lahore
```

```
In [7]: #Open a file to append
        fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","a+")
        fileOpen.write("\n New Here with new line")
        fileOpen.close()

        #open a file to read
        fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","r+")
        string = fileOpen.read() #to read specific content from start you can use read(12). It
        print(string)
        #close opend file
        fileOpen.close()
```

```
Name: Mehvish Ashiq
Department: BSCS
```

```
 Computer Science
 Computer Science Information Technology Lahore
 New Here with new line
```

```
In [9]: fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","w")

        fileOpen.close()

        fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","a+")
        fileOpen.write("Khalil Ul Rehman \n Software Engineer")
        fileOpen.close()

        #open a file to read
        fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","r+")
        string = fileOpen.read() #to read specific content from start you can use read(12). It
        print(string)
        #close opend file
        fileOpen.close()
```

```
Khalil Ul Rehman
 Software Engineer
```

```
In [10]: fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","a+")
         fileOpen.write("\nInstitute COMSATS")
         fileOpen.close()

         #open a file to read
         fileOpen = open("H:/RCS/Semester 2/Machine Learning/Assignment 1/file.txt","r+")
         string = fileOpen.read() #to read specific content from start you can use read(12). I
         print(string)
         #close opend file
         fileOpen.close()
```

```
Khalil Ul Rehman
 Software Engineer
Institute COMSATS
```

8.3 8.3 File Position

- tell() tells the current position within the file.
- seeks() method changes the current file location.
- With os we can rename and remove file

```
In [11]: # Open a file
         fo = open("file.txt","r+")
         str = fo.read(10)
```

```
        print("Read String is : \n",str)

        #check current position
        position = fo.tell()
        print("Current file position : \n",position)

        #Reposition pointer at the beginning once again
        position = fo.seek(0,0)
        str = fo.read(10)
        print("Again read String is : \n",str)
        #close open file
        fo.close()
```

```
Read String is :
  Informati
Current file position :
 10
Again read String is :
  Informati
```

```
In [17]: fileOpen = open("file.txt","r+")
         stringValue = fileOpen.read()
         print(stringValue,"Location :",fileOpen.tell())

         fileOpen.seek(5,0)
         fileOpen.write(": New Text :")
         fileOpen.close()

         fileOpen = open("file.txt","r+")
         print(fileOpen.read())

         fileOpen.close()
```

```
 Info: New Text :nology Lahore Information Technology Lahore Location : 60
 Info: New Text :nology Lahore Information Technology Lahore
```

```
In [18]: fileOpen = open("file.txt","r+")
         stringValue = fileOpen.read(2)
         print("Location :",fileOpen.tell())
         stringValue = fileOpen.read(2)
         print("Location :",fileOpen.tell())
         stringValue = fileOpen.read(2)
         print("Location :",fileOpen.tell())
         fileOpen.close()
```

```
Location : 2
Location : 4
```

```
Location : 6
```

```
In [19]: fileOpen = open("file.txt","r+")
         stringValue = fileOpen.read(2)
         fileOpen.seek(2,0)
         fileOpen.write(": We are writing after seek :")
         fileOpen.seek(2,0)
         fileOpen.write(": We are writing after seek :")
         fileOpen.close()

         fileOpen = open("file.txt","r+")
         print(fileOpen.read())
         fileOpen.close()
```

```
 I: We are writing after seek :Information Technology Lahore
```

```
In [22]: import os
         #rename a file
         os.rename("file.txt","newfile.txt")
```

```
In [23]: import os
         #remove file
         os.remove("newfile.txt")
```

9 9 Pandas Introduction

- Pandas is an open source library built on top of NumPy
- It allows for fast analysis and data cleaning and preparation
- It excels in proformance and productivity
- It also has built-in visualization features
- It can work with data from a wide variety of sources

10 10 Series

- A series is very similar to NumPy array.
- Series is 1-D array labeled array capable of holding any type of data.
- The difference between the NumPy array from a Series, is that a Series can have axis labels, meaning it can indexed by a label insted of just a number location.
- The axis labels are collectively referred to as the index.
- Following function is used to create a series: `s = pd.Series(data,index=index)`
- In above function, data can be many different things:
  - A Python dict
  - An ndarray
  - A scalar value (For example : 5)

- The passed index is a list of axis labels. So, this seperates into a few cases depending on what data is:
- Operations between Series (+, -, /, , *) align values based on their associated index values– they need not be the same length. The result index will be the sorted union of the two indexes.

Why

```
To save a 1-D array fro visualization or manuplation.
```

When

```
We use it when we need to mannuplation and visualization of 1-D array.
```

Where

```
We can can create it in any code block.
```

10.1 10.1 From ndarray

- If data is an ndarray, index must be the same length as data
- If no index is passed, one will be created having values `[0,...,len(data)-1]`

For understanding examples are given below.

```
In [29]: import pandas as pd
         import numpy as np
         import matplotlib.pyplot as plt

         # following a function is called from pandas to create a series. Data would be
         # 5 random values and indexess are assigned a-e

         s = pd.Series(np.random.randn(5),index = ['a','b','c','d','e'])
         s

Out[29]: a    0.607894
         b    0.939601
         c   -0.117312
         d    0.871365
         e    0.299762
         dtype: float64

In [32]: # Following function will print the index and its datatype
         s.index

Out[32]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [34]: # If we will not assign the index then it will of length having values [0...Len(data-
         pd.Series(np.random.randn(5))
```

```
Out[34]: 0    0.619704
         1   -0.806142
         2   -0.455169
         3   -2.149248
         4    0.187618
         dtype: float64
```

```
In [47]: import pandas as pd
         import numpy as np

         series = pd.Series( np.random.bytes(10))
         series
```

```
Out[47]: 0    b'\x08D\x95\x03\x9b\xd8\xa8\x1c?\xda'
         dtype: object
```

```
In [48]: series_integers = pd.Series(np.random.randint(5, size = 10))
         series_integers
```

```
Out[48]: 0    0
         1    4
         2    4
         3    2
         4    3
         5    1
         6    1
         7    0
         8    1
         9    2
         dtype: int32
```

```
In [50]: series = pd.Series(np.random.choice(500,10))
         series
```

```
Out[50]: 0    127
         1    425
         2    176
         3    336
         4     39
         5     94
         6    221
         7    396
         8    446
         9     94
         dtype: int32
```

10.2 10.2 From dict

- If data is a dict, if index is passed the values in data corresponding to the labels in the index will be pulled out.

- If index is not passed then it will be constructed from the sorted keys of the dict, if possible.

```
In [51]: # In following example, indexs are not given to it is constructed from the
         # sorted key of the dict
         d = {'a':0.,'b':1.,'c':2.} # a python dict
         pd.Series(d)

Out[51]: a    0.0
         b    1.0
         c    2.0
         dtype: float64

In [53]: # in the following example, index are given, so the values in data
         # corresponding in the index will be pulled out
         pd.Series(d,index = ['b','c','d','a'])

Out[53]: b    1.0
         c    2.0
         d    NaN
         a    0.0
         dtype: float64

In [54]: d = {'89':1,'30':500,'65':34}
         pd.Series(d)

Out[54]: 89      1
         30    500
         65     34
         dtype: int64

In [55]: d = {'c':1,'a':500,'b':34}
         pd.Series(d)

Out[55]: c      1
         a    500
         b     34
         dtype: int64

In [58]: d = {'c':1,'a':500,'b':34}
         pd.Series(d,index=[1,500,'a',2,56])

Out[58]: 1        NaN
         500      NaN
         a      500.0
         2        NaN
         56       NaN
         dtype: float64
```

10.3 10.3 From a Scalar Value

- If data is a scalar value, an index must be provided. The value will be repeated to match the length of index

```
In [59]: # in the following example, a scalar value is given as data so it
         # will be repeated to match the length of index
         pd.Series(5., index= ['a','b','c','d','e'])
```

```
Out[59]: a    5.0
         b    5.0
         c    5.0
         d    5.0
         e    5.0
         dtype: float64
```

```
In [60]: pd.Series('Khalil',index = [1,3,5,6])
```

```
Out[60]: 1    Khalil
         3    Khalil
         5    Khalil
         6    Khalil
         dtype: object
```

```
In [61]: pd.Series(index = [1,2,3,4,5,6])
```

```
Out[61]: 1   NaN
         2   NaN
         3   NaN
         4   NaN
         5   NaN
         6   NaN
         dtype: float64
```

```
In [62]: pd.Series(555,['first','secound','third','fourth'])
```

```
Out[62]: first      555
         secound    555
         third      555
         fourth     555
         dtype: int64
```

10.4 10.4 Series is ndarray-like

- It acts very similarly to a ndarray.
- It is a valid argument to most NumPy functions. However, things like slicing also slice the index.

```
In [63]: # We can access a value just like ndarray
         # access single value
         s[0]
```

```
Out[63]: 0.6078942359551488

In [64]: #access range of values
         s[:5]

Out[64]: a    0.607894
         b    0.939601
         c   -0.117312
         d    0.871365
         e    0.299762
         dtype: float64

In [65]: #Following example will return a range of values in seriess whose value is
         # greater than the median of series
         s[s > s.median()]

Out[65]: b    0.939601
         d    0.871365
         dtype: float64

In [66]: # Following example is return the values in series with indexes. 4,3,,1 are
         # the positions of the indexes For Example: the index at 4,3,1 are edb respectively
         s[[4,3,1]]

Out[66]: e    0.299762
         d    0.871365
         b    0.939601
         dtype: float64

In [67]: # Following example returns the exponent values. Just like e^a
         #  (here a is index and its respective data is placed here)
         np.exp(s)

Out[67]: a    1.836560
         b    2.558961
         c    0.889308
         d    2.390171
         e    1.349538
         dtype: float64

In [68]: # Following Example will gett the data of given index
         s['a']

Out[68]: 0.6078942359551488

In [70]: # Following example will update the data of the given index
         s['e'] = 12
         s # before updating e = 1.349538 and after update it will be 12.000000
```

```
Out[70]: a     0.607894
         b     0.939601
         c    -0.117312
         d     0.871365
         e    12.000000
         dtype: float64

In [71]: # Following will return true if 'e' is in the values of index otherwise false
         'e' in s

Out[71]: True

In [72]: # If a lable is not contained and you are trying to access its data, an
         # exception will raise
         s['f'] #this will create error


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         c:\program files (x86)\python37-32\lib\site-packages\pandas\core\indexes\base.py in ge
        3123             try:
     -> 3124                 return libindex.get_value_box(s, key)
        3125             except IndexError:


         pandas\_libs\index.pyx in pandas._libs.index.get_value_box()


         pandas\_libs\index.pyx in pandas._libs.index.get_value_box()


         TypeError: 'str' object cannot be interpreted as an integer


     During handling of the above exception, another exception occurred:


         KeyError                                  Traceback (most recent call last)

         <ipython-input-72-6a105faac945> in <module>()
           1 # If a lable is not contained and you are trying to access its data, an
           2 # exception will raise
     ----> 3 s['f'] #this will create error


         c:\program files (x86)\python37-32\lib\site-packages\pandas\core\series.py in __getite
         765         key = com._apply_if_callable(key, self)
```

```
    766         try:
--> 767             result = self.index.get_value(self, key)
    768
    769             if not is_scalar(result):


    c:\program files (x86)\python37-32\lib\site-packages\pandas\core\indexes\base.py in get
   3130                     raise InvalidIndexError(key)
   3131                 else:
-> 3132                     raise e1
   3133             except Exception:  # pragma: no cover
   3134                 raise e1


    c:\program files (x86)\python37-32\lib\site-packages\pandas\core\indexes\base.py in get
   3116         try:
   3117             return self._engine.get_value(s, k,
-> 3118                                           tz=getattr(series.dtype, 'tz', None))
   3119         except KeyError as e1:
   3120             if len(self) > 0 and self.inferred_type in ['integer', 'boolean']:


    pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()


    pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()


    pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()


    pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get


    pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get


    KeyError: 'f'
```

```
In [75]: # Using the get method, a missing lable will return None or specified default
         s.get('f') # it will return none
         s.get('f', np.nan) # it will return default value
```

```
Out[75]: nan
```

```
In [2]: import pandas as pd
        series = pd.Series({'a':'a','b':'b','c':'c','d':'d'})
        series[:2]
```

```
Out[2]: a    a
        b    b
        dtype: object
```

```
In [4]: series[['c','d','a']]
```

```
Out[4]: c    c
        d    d
        a    a
        dtype: object
```

```
In [5]: series['a'] = 500
        series
```

```
Out[5]: a    500
        b      b
        c      c
        d      d
        dtype: object
```

10.5 10.6 Vectorized Operations and Label Alignment with Series

- When doing data analysis, as with raw NumPy arrays looping through Series value-by-value is usally not necessary.
- Series can also be passed into most NumPy methods expecting an ndarray.

```
In [8]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt


        s = pd.Series(np.random.randn(5),index = ['a','b','c','d','e'])
        s['e'] = 12

        # Following will add the data of respective values of indexes. For example,
        # in given output, it is calculate d as:
        a = s['a'] + s['a']
        b = s['b'] + s['b']
        c = s['c'] + s['c']
        d = s['d'] + s['d']
        e = s['e'] + s['e']

        s+s
```

```
Out[8]: a    -1.798753
        b    -0.162378
        c     2.543501
        d     0.604762
        e    24.000000
        dtype: float64
```

```
In [9]: # Following will multiply the data of each values of indexes, with 2.
        # For example, in given output, it is calculated as:
        a = s['a'] *2
        b = s['b'] *2
        c = s['c'] *2
        d = s['d'] *2
        e = s['e'] *2

        s*2

Out[9]: a    -1.798753
        b    -0.162378
        c     2.543501
        d     0.604762
        e    24.000000
        dtype: float64

In [10]: s = pd.Series(np.random.randn(5), name = 'something')
         s

Out[10]: 0   -1.003962
         1   -0.650802
         2   -0.781763
         3   -0.367287
         4    0.273567
         Name: something, dtype: float64

In [11]: s.name #print the name attribute of series

Out[11]: 'something'

In [12]: #rename the series name attribute and assign to s2 object. Note that s and s2
         # refers to different objects.
         s2 = s.rename('different')
         s2.name

Out[12]: 'different'

In [20]: series = pd.Series(555,['first','secound','third','fourth'])
         series['secound']
         series['secound'] = series['secound'] * 10
         series

Out[20]: first       555
         secound    5550
         third       555
         fourth      555
         dtype: int64

In [21]: series + series
```

```
Out[21]: first       1110
         secound    11100
         third       1110
         fourth      1110
         dtype: int64

In [22]: series * series

Out[22]: first        308025
         secound    30802500
         third        308025
         fourth       308025
         dtype: int64

In [23]: series.name = "Series of Integers"
         series.name

Out[23]: 'Series of Integers'

In [26]: s = series.rename('another')
         s.name
         third = s['third'] / 5000
         s / 5000

Out[26]: first      0.111
         secound    1.110
         third      0.111
         fourth     0.111
         Name: another, dtype: float64

In [27]: s = s*s / 10000
         s

Out[27]: first        30.8025
         secound    3080.2500
         third        30.8025
         fourth       30.8025
         Name: another, dtype: float64
```

11 11 Data Frames

- DataFrames are the workhorse of the pandas and are directly inspired by the R programming language.
- Each row in your data frame represents a data sample.
- Like Series, DataFrame accepts many different kinds of input:
  - Dict of 1D ndarrays, lists, dicts, or Series
  - 2-D numpy.ndarray
  - Structured or record ndarray

- A Series
- Another DataFrame
  * Along with the data, you can optionally pass index (row labela) and columns (column labels) arguments.
  * If you pass an index and/or columns, you are guaranteeing the index and/or columns of the resulting DataFrame.
  * Thus, a dict of series plus a specific index will discard all data not matching up to the passed index.
  * If axis labels are not passed, they will be constructed from the input data based on common sense rules.

Why

```
To visualize and prepross 1-D and 2-D arrays.
```

When

```
We use it when we need to prepross and visualize 1-D and 2-D arrays.
```

Where

```
We can can create it in any code block.
```

11.1 11.1 From dict of Series or dicts

- The result index will be the union of the indexes of the various Series.
- If there are any nested dicts, there will be first converted to Series.
- If no columns are passed, the columns will be the sorted list of dict keys.

```
In [28]: # A dict is created
         d = {
             'one' : pd.Series([1.,2.,3.],index = ['a','b','c']),
             'two' : pd.Series([1.,2.,3.,4.] , index = ['a','b','c','d'])
         }

         # crete a dataframe. row label will be the index of a series. As
         # coloum labels are not given so it will be sorted list of dict keys

         df = pd.DataFrame(d)
         df

Out[28]:    one  two
         a  1.0  1.0
         b  2.0  2.0
         c  3.0  3.0
         d  NaN  4.0

In [29]: # a data frame will be constructed for given row labels
         pd.DataFrame(d, index = ['d','b','a'])
```

```
Out[29]:    one  two
          d  NaN  4.0
          b  2.0  2.0
          a  1.0  1.0

In [30]: # Following example shows a data frame when we give column labels
         pd.DataFrame(d, index = ['d','b','a'], columns = ['two','three'])

Out[30]:    two three
          d  4.0   NaN
          b  2.0   NaN
          a  1.0   NaN

In [31]: df.columns

Out[31]: Index(['one', 'two'], dtype='object')

In [42]: data = {
             'weight': pd.Series('40kg', index = [1,2,3,4,5,6,7,8,9]),
             'age': pd.Series(np.random.randint(15,50),index = [1,2,3,4,5,6,7,8,9])
         }
         pd.DataFrame(data , columns=['weight','age'])

Out[42]:   weight  age
          1   40kg   19
          2   40kg   19
          3   40kg   19
          4   40kg   19
          5   40kg   19
          6   40kg   19
          7   40kg   19
          8   40kg   19
          9   40kg   19

In [44]: data = {
             'name' : pd.Series(['Khalil','Usman','Zaneb','Ruqail','Mehvish']),
             'weight': pd.Series('40kg', index = [1,2,3,4,5,6,7,8,9]),
             'age': pd.Series(np.random.randint(15,50),index = [1,2,3,4,5,6,7,8,9])
         }
         pd.DataFrame(data , columns=['weight','age','name'])

Out[44]:   weight   age     name
          0    NaN   NaN   Khalil
          1   40kg  17.0    Usman
          2   40kg  17.0    Zaneb
          3   40kg  17.0   Ruqail
          4   40kg  17.0  Mehvish
          5   40kg  17.0      NaN
          6   40kg  17.0      NaN
```

```
7   40kg  17.0      NaN
8   40kg  17.0      NaN
9   40kg  17.0      NaN
```

```
In [45]: data = {
             'name' : pd.Series(['p1','p2','p3','p4','p5','p6']),
             'weight': pd.Series('40kg', index = [1,2,3,4,5,6]),
             'price': pd.Series(np.random.randint(15,50),index = [1,2,3,4,5,6])
         }
         pd.DataFrame(data )
```

```
Out[45]:   name weight  price
         0   p1    NaN    NaN
         1   p2   40kg   49.0
         2   p3   40kg   49.0
         3   p4   40kg   49.0
         4   p5   40kg   49.0
         5   p6   40kg   49.0
         6  NaN   40kg   49.0
```

11.2 11.2 From dict of ndarrays/lists

- The ndarrays must all be the same length.
- If an index is passed, it must clearly also be the same length as the arrays.
- If no index is passed, the result will be range(n), where n is the array length.

```
In [46]: # Following examples shows that ndarray has same length
         d = {
             'one' : [1.,2.,3.,4.],
             'two' : [4.,3.,2.,1.]
         }
         #column labels are not given so the result will be range(n),
         # Where n is the array length
         pd.DataFrame(d)
```

```
Out[46]:    one  two
         0  1.0  4.0
         1  2.0  3.0
         2  3.0  2.0
         3  4.0  1.0
```

```
In [47]: #If indexs are given then it would be same length as arrays
         pd.DataFrame(d , index = ['a','b','c','d'])
```

```
Out[47]:    one  two
         a  1.0  4.0
         b  2.0  3.0
         c  3.0  2.0
         d  4.0  1.0
```

```
In [48]: data = {
             'weight': [12,12,5,6,85],
             'age': [55,56,51,58,52]
         }
         pd.DataFrame(data )
```

Out[48]:

	weight	age
0	12	55
1	12	56
2	5	51
3	6	58
4	85	52

```
In [49]: pd.DataFrame(data, index = ['one','two','three','four','five'])
```

Out[49]:

	weight	age
one	12	55
two	12	56
three	5	51
four	6	58
five	85	52

```
In [56]: data = {
             'name' : ['p1','p2','p3','p4','p5','p6'],
             'weight': [45,545,54,5,95,89],
             'price': [58,85,900,650,1569,96]
         }
         pd.DataFrame(data, index = [500,200,300,400,50,63])
```

Out[56]:

	name	weight	price
500	p1	45	58
200	p2	545	85
300	p3	54	900
400	p4	5	650
50	p5	95	1569
63	p6	89	96

```
In [58]: data = {
             'series1' : [1,2,3,4,5,6,7,8,9],
             'series2' : [1,4,9,16,25,36,49,64,81],
             'series3' : [0,1,2,3,4,5,6,7,8]
         }
         pd.DataFrame(data)
```

Out[58]:

	series1	series2	series3
0	1	1	0
1	2	4	1
2	3	9	2
3	4	16	3

```
4        5       25        4
5        6       36        5
6        7       49        6
7        8       64        7
8        9       81        8
```

```
In [59]: pd.DataFrame(data, index = ['1','8','6','6','9','52','8','2','23'])
```

```
Out[59]:     series1  series2  series3
         1         1        1        0
         8         2        4        1
         6         3        9        2
         6         4       16        3
         9         5       25        4
         52        6       36        5
         8         7       49        6
         2         8       64        7
         23        9       81        8
```

11.3 11.3 From a list of dicts

```
In [61]: # Constructing data frame from a list of dicts
         data2 = [{'a':1,'b':2},{'a':5,'b':10,'c':20}]
         pd.DataFrame(data2)
```

```
Out[61]:    a   b     c
         0  1   2   NaN
         1  5  10  20.0
```

```
In [62]: #passing list of dicts as data and indexes (row labels)
         pd.DataFrame(data2, index = ['First','Secound'])
```

```
Out[62]:          a   b     c
         First    1   2   NaN
         Secound  5  10  20.0
```

```
In [63]: # Passing list of dicts as data and columns (columns labels)
         pd.DataFrame(data2,columns = ['a','b'])
```

```
Out[63]:    a   b
         0  1   2
         1  5  10
```

```
In [64]: my_data = [{'Name':'Khalil','Age':23,'Weight':75},
                    {'Name':'Zahid','Age':29,'Weight':85},
                    {'Name':'Shumail','Age':23,'Weight':45}]
         pd.DataFrame(my_data)
```

```
Out[64]:    Age     Name  Weight
         0   23   Khalil      75
         1   29    Zahid      85
         2   23  Shumail      45
```

```
In [65]: my_data = [{'Name':'Khalil','Age':23,'Weight':75},
                    {'Name':'Zahid','Age':29,'Weight':85},
                    {'Name':'Shumail','Age':23,'Weight':45}]
         pd.DataFrame(my_data, index = [1,2,33])
```

```
Out[65]:      Age     Name  Weight
         1     23   Khalil      75
         2     29    Zahid      85
         33    23  Shumail      45
```

```
In [66]: my_data = [{'Name':'Khalil','Age':23,'Weight':75},
                    {'Name':'Zahid','Age':29,'Weight':85},
                    {'Name':'Shumail','Age':23,'Weight':45}]
         pd.DataFrame(my_data, columns = ['Name','Age','Weight'])
```

```
Out[66]:       Name  Age  Weight
         0   Khalil   23      75
         1    Zahid   29      85
         2  Shumail   23      45
```

11.4 11.4 From a dict of tuples

You can automatically create a multi-indexed frame by passing a tuples dictionary

```
In [67]: pd.DataFrame({
             ('a','b'):{('A','B'):1, ('A','C'):2},
             ('a','a'):{('A','C'):3, ('A','B'):4},
             ('a','c'):{('A','B'):5, ('A','C'):6},
             ('b','a'):{('A','C'):7, ('A','B'):8},
             ('b','b'):{('A','D'):9, ('A','B'):10}
         })
```

```
Out[67]:         a              b
                 b    a    c    a     b
         A B   1.0  4.0  5.0  8.0  10.0
           C   2.0  3.0  6.0  7.0   NaN
           D   NaN  NaN  NaN  NaN   9.0
```

```
In [2]: import pandas as pd
        pd.DataFrame({
            ('Type1','1'):{('Direction X','Length'):1,('Direction X','Width'):5},
            ('Type2','1'):{('Direction X','Length'):5,('Direction X','Width'):8}
        })
```

```
Out[2]:                      Type1 Type2
                                 1     1
        Direction X Length       1     5
                    Width        5     8
```

```
In [4]: import pandas as pd
        pd.DataFrame({
            ('Face1','A'):{('Span X','Length'):1,('Span X','Width'):5,('Span Y','Length'):29,(
            ('Face2','B'):{('Span X','Length'):5,('Span X','Width'):8}
        })
```

```
Out[4]:                Face1 Face2
                           A     B
        Span X Length      1   5.0
               Width       5   8.0
        Span Y Length     29   NaN
               Width      53   NaN
```

11.5 11.5 Alternate Constructors

DataFrame.from_dict * **DataFrame.from_dict** takes a dict of dicts or a dict of array-like sequences and returns a **DataFrame**. * It operates like the DataFrame constructor except for the orient parameter which is 'columns' by default, but which can be set to 'index' in order to use the dict keys as row labels.

DataFrame.from_records

- **DataFrame.from_records** takes a list of tupless or an ndarray with structured dtype.
- Works analogously to the normal DataFrame constructor, except that index maybe be a specific field of the structured dtype to use as the index. For example:

```
In [6]: import numpy as np
        data = np.zeros((2,), dtype = [('A','i4'),('B','f4'),('C','a10')])
        data
```

```
Out[6]: array([(0, 0., b''), (0, 0., b'')],
              dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
```

```
In [7]: pd.DataFrame.from_records(data, index='C')
```

```
Out[7]:      A    B
        C
        b''  0  0.0
        b''  0  0.0
```

```
In [14]: data = [(4,5,6),(58,95,58)]
         data
```

```
Out[14]: [(4, 5, 6), (58, 95, 58)]
```

```
In [15]: pd.DataFrame.from_records(data)
```

```
Out[15]:     0   1   2
         0   4   5   6
         1  58  95  58
```

DataFrame.from_items * **DataFrame.from_items** works analogously to the form of the dict constructor that takes a sequence of (key, value) pairs, where the keys are column (or row, in the case of orient='index') names, and the value are the column values (or row values). * This can be useful for constructing a DataFrame with the columns in a particular order without having to pass an explicit list of columns

```
In [18]: pd.DataFrame.from_items([('A',[1,2,3]),('B',[4,5,6])])

c:\program files (x86)\python37-32\lib\site-packages\ipykernel_launcher.py:1: FutureWarning: fr
  """Entry point for launching an IPython kernel.


Out[18]:    A  B
         0  1  4
         1  2  5
         2  3  6

In [20]: pd.DataFrame.from_items([('A',[1,2,3]),('B',[4,5,6])], orient='index',columns=['one',

c:\program files (x86)\python37-32\lib\site-packages\ipykernel_launcher.py:1: FutureWarning: fr
  """Entry point for launching an IPython kernel.


Out[20]:    one  two  three
         A    1    2      3
         B    4    5      6
```

11.6 11.6 Column selection, addition, deletion

- DataFraame can be treated semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntex as the analogous dict operations.

```
In [2]: import pandas as pd
        df = pd.DataFrame.from_items([('A',[1,2,3]),('B',[4,5,6])], orient='index',columns=['on
        df['one'] #it is displaying data under coloumn 'one'

c:\program files (x86)\python37-32\lib\site-packages\ipykernel_launcher.py:2: FutureWarning: fr


Out[2]: A    1
        B    4
        Name: one, dtype: int64

In [35]: df['three'] = df['one'] * df['two']  #assigning values to a colomn named 'three' afte
         df

Out[35]:    one  two  three
         A    1    2      2
         B    4    5     20
```

```
In [36]: df['flag'] = df['one'] > 2 # check if value at column 'one' is > 2 then assign True o
         df

Out[36]:    one  two  three   flag
         A    1    2      2  False
         B    4    5     20   True
```

Columns, can be deleted or popped like with a dict:

```
In [37]: del df['two'] # delete a column 'two' from data frame

In [38]: three = df.pop('three') #pop a complete column 'three' from data frame

In [39]: df

Out[39]:    one   flag
         A    1  False
         B    4   True
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [40]: df['foo'] = 'bar' #a column 'foo' will be populated with 'bar'

In [41]: df

Out[41]:    one   flag  foo
         A    1  False  bar
         B    4   True  bar
```

When inserting a Series that does not have the same index as the DataFrame, It will be conformed to the DataFrame's index

```
In [42]: #Following example will take values from column one until given range
         # and will populate the new column
         df['one_trunc'] = df['one'][:1]
         df

Out[42]:    one   flag  foo  one_trunc
         A    1  False  bar        1.0
         B    4   True  bar        NaN
```

By default, colun get inserted at the end . The insert function is available to insert at a particular ocation in the columns:

```
In [3]: #Following function has three arguments.
        #First argument: inex where new column will be inserted.
        #Secound argument: label or title of a new coloumn
        #Third argument: It will create a coloumn at specified position
        df.insert(1,'bar2',df['one'])

In [4]: df
```

```
Out[4]:    one  bar2  two  three
        A    1     1    2      3
        B    4     4    5      6

In [7]: import pandas as pd
        df = pd.DataFrame.from_items([('First',[100,'Khalil']),('Secound',[105,'Junaid']),('Th

c:\program files (x86)\python37-32\lib\site-packages\ipykernel_launcher.py:2: FutureWarning: fr



In [8]: df

Out[8]:          Numbers   names
        First        100  Khalil
        Secound      105  Junaid
        Third        165   Usman

In [9]: df['money'] = df['Numbers'] * 100
        df

Out[9]:          Numbers   names  money
        First        100  Khalil  10000
        Secound      105  Junaid  10500
        Third        165   Usman  16500

In [10]: df['flag'] = False

In [11]: df

Out[11]:          Numbers   names  money   flag
         First        100  Khalil  10000  False
         Secound      105  Junaid  10500  False
         Third        165   Usman  16500  False

In [12]: del df['flag']
         df

Out[12]:          Numbers   names  money
         First        100  Khalil  10000
         Secound      105  Junaid  10500
         Third        165   Usman  16500

In [13]: df.pop('Numbers')
         df

Out[13]:           names  money
         First    Khalil  10000
         Secound  Junaid  10500
         Third     Usman  16500
```

```
In [14]: df.insert(0,'Numbers',[5,6,89])
         df

Out[14]:          Numbers   names  money
         First          5  Khalil  10000
         Secound        6  Junaid  10500
         Third         89   Usman  16500
```

11.7 11.7 Indexing/Selection

- Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [20]: df.loc['Secound'] #it will return the coloumn lables and values on row label 'b'

Out[20]: Numbers         6
         names      Junaid
         money       10500
         Name: Secound, dtype: object

In [21]: df.iloc[2] #it will return the values of those coloumns that is > than 2

Out[21]: Numbers       89
         names      Usman
         money      16500
         Name: Third, dtype: object

In [24]: df.iloc[0,2]

Out[24]: 10000
```

If we will pass the index and column then we will get the respected value of crossection of index and column

```
In [25]: df.loc['Secound','names']

Out[25]: 'Junaid'

In [26]: df.head(5)

Out[26]:          Numbers   names  money
         First          5  Khalil  10000
         Secound        6  Junaid  10500
         Third         89   Usman  16500
```

If we will pass the indexs then we will get only the data that is available on that index.

```
In [27]: df.loc[['Secound','Third']]

Out[27]:          Numbers   names  money
         Secound        6  Junaid  10500
         Third         89   Usman  16500
```

11.8 11.8 Data Alignment and Arithmetic

- Data alignment between DataFrame objects automatically align on both the columns and the index (row labels).
- Again, the resulting object will have the union of the column and row labels.

```
In [32]: import numpy as np
         df = pd.DataFrame(np.random.randn(10,4), columns=['A','B','C','D'])

In [33]: df2 = pd.DataFrame(np.random.randn(7,3), columns=['A','B','C'])

In [34]: df + df2 #add values of respectie column labels
```

Out[34]:

	A	B	C	D
0	0.132694	1.490645	0.608474	NaN
1	0.048412	2.145690	0.445568	NaN
2	-1.377930	-1.454274	1.892880	NaN
3	1.564354	0.140908	-0.015433	NaN
4	0.107471	-1.516738	0.840583	NaN
5	1.673640	0.106717	1.750814	NaN
6	-0.583583	0.491956	-0.015101	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN

```
In [35]: df - df.iloc[0]
```

Out[35]:

	A	B	C	D
0	0.000000	0.000000	0.000000	0.000000
1	1.061286	-0.412053	-0.336569	-2.303392
2	0.151297	-1.902361	1.359637	-1.836160
3	2.010503	-1.815079	-0.555547	0.547968
4	0.329896	-1.817588	-0.517536	-2.551260
5	1.799843	-0.652418	0.451955	-0.002412
6	-0.053599	-0.166202	-0.788508	-0.288498
7	2.323315	-1.462654	-1.734110	-2.306665
8	-0.070894	-0.244438	-0.441285	0.065661
9	-0.389353	-0.529104	-1.155666	-2.164832

```
In [36]: df * 5 + 2
```

Out[36]:

	A	B	C	D
0	-0.810416	8.816446	3.552818	7.705845
1	4.496016	6.756181	1.869971	-3.811113
2	-0.053931	-0.695357	10.351004	-1.474954
3	9.242100	-0.258951	0.775083	10.445686
4	0.839066	-0.271494	0.965137	-5.050454
5	8.188799	5.554357	5.812594	7.693784
6	-1.078413	7.985437	-0.389723	6.263355
7	10.806159	1.503177	-5.117734	-3.827479
8	-1.164885	7.594258	1.346393	8.034152
9	-2.757183	6.170926	-2.225513	-3.118315

```
In [37]: 1 / df
```

```
Out[37]:           A          B          C         D
         0 -1.779096   0.733520   3.219953  0.876294
         1  2.003192   1.051264 -38.453048 -0.860420
         2 -2.434357  -1.855042   0.598730 -1.438868
         3  0.690407  -2.213416  -4.081908  0.592018
         4 -4.306876  -2.201194  -4.831558 -0.709174
         5  0.807911   1.406724   1.311443  0.878151
         6 -1.624214   0.835361  -2.092293  1.172785
         7  0.567784 -10.063948  -0.702471 -0.858004
         8 -1.579836   0.893774  -7.649862  0.828617
         9 -1.051042   1.198775  -1.183288 -0.976884
```

```
In [38]: df ** 4
```

```
Out[38]:           A         B             C         D
         0  0.099817  3.454235  9.302547e-03  1.695899
         1  0.062103  0.818754  4.573798e-07  1.824556
         2  0.028475  0.084447  7.781703e+00  0.233301
         3  4.401270  0.041663  3.602030e-03  8.140662
         4  0.002906  0.042596  1.835064e-03  3.953560
         5  2.347175  0.255367  3.380667e-01  1.681605
         6  0.143690  2.053542  5.218070e-02  0.528598
         7  9.622014  0.000097  4.106643e+00  1.845198
         8  0.160528  1.567075  2.920019e-04  2.121216
         9  0.819445  0.484228  5.100792e-01  1.098065
```

Boolean operators work as well:

```
In [39]: df1 = pd.DataFrame({'a':[1,0,1],'b':[0,1,1]}, dtype=bool)
```

```
In [40]: df2 = pd.DataFrame({'a':[0,1,1],'b':[1,1,0]}, dtype=bool)
```

```
In [41]: pd.DataFrame({'a':[0,1,1],'b':[1,1,0]}, dtype=bool)
```

```
Out[41]:        a      b
         0  False   True
         1   True   True
         2   True  False
```

```
In [42]: df1 & df2 #and logical operator
```

```
Out[42]:        a      b
         0  False  False
         1  False   True
         2   True  False
```

```
In [43]: df1 | df2 # or operator
```

```
Out[43]:       a     b
         0  True  True
         1  True  True
         2  True  True

In [44]: -df1

Out[44]:       a      b
         0  False   True
         1   True  False
         2  False  False

In [45]: df1['a'] = True
         df1

Out[45]:       a      b
         0  True  False
         1  True   True
         2  True   True

In [46]: df1 = df1 & df1 | df1

In [47]: df1

Out[47]:       a      b
         0  True  False
         1  True   True
         2  True   True

In [48]: df1 = -df1

In [49]: df1

Out[49]:       a      b
         0  False   True
         1  False  False
         2  False  False
```

11.9 11.9 Transposing

- To transpose, access the T attribute (also the transpose function), similar to an ndarray

```
In [50]: #only show the first 5 rows
         df[:5].T

Out[50]:           0         1         2         3         4
         A -0.562083  0.499203 -0.410786  1.448420 -0.232187
         B  1.363289  0.951236 -0.539071 -0.451790 -0.454299
         C  0.310564 -0.026006  1.670201 -0.244983 -0.206973
         D  1.141169 -1.162223 -0.694991  1.689137 -1.410091
```

Creating a DataFrame by passing a numpy array, with a datetime index and labeled column:

```
In [51]: dates = pd.date_range('20130101',periods=6)
         dates

Out[51]: DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
                        '2013-01-05', '2013-01-06'],
                       dtype='datetime64[ns]', freq='D')

In [52]: df = pd.DataFrame(np.random.randn(6,4), index = dates, columns = list('ABCD'))
         df

Out[52]:                    A         B         C         D
         2013-01-01 -0.287586 -2.112358  0.526925  0.882804
         2013-01-02  0.137791  0.786193 -1.043605  0.316305
         2013-01-03  0.538533 -0.059857 -0.212885 -0.005387
         2013-01-04 -0.896319  1.084214 -0.023529  1.932443
         2013-01-05 -0.144685 -0.741743  1.695067  0.689757
         2013-01-06  0.872990  0.804680  0.309702 -0.162149
```

Creating a DataFrame by passing a dict of objects thaat can be converted to series-like

```
In [55]: df2 = pd.DataFrame({
             'A':1.,
             'B':pd.Timestamp('20130102'),
             'C':pd.Series(1,index=list('1234'),dtype='float32'),
             'D':np.array([3]*4,dtype='int32'),
             'E':pd.Categorical(["test","train","test","train"]),
             'F':'foo'
         })
         df2

Out[55]:      A          B    C  D      E    F
         1  1.0 2013-01-02  1.0  3   test  foo
         2  1.0 2013-01-02  1.0  3  train  foo
         3  1.0 2013-01-02  1.0  3   test  foo
         4  1.0 2013-01-02  1.0  3  train  foo

In [56]: #Having specific dtype
         df2.dtypes

Out[56]: A           float64
         B    datetime64[ns]
         C           float32
         D             int32
         E          category
         F            object
         dtype: object
```

12 12 Viewing Data

- We can view data/display data in different ways:
- See the top & bottom rows of the frame
- Selecting a single column
- Selecting via [], which slices the rows
- For getting a cross section using a label
- Selecting on a multi-axis by label
- Showing lable slicing, both endpoints are included
- Reduction in the dimensions of the returned object
- For getting a scalar value
- For getting fast access to a scalar
- Select via the position of the passed integers
- By interger slices, acting similar to numpy/python
- By lists of integer position Locations, similar to the numpy/python style
- For slicing rows explicitly
- For slicing columns explicitly
- For getting a value explicitly
- For getting fast access to a scalar
- Using a single columns values to select data
- Selecting values from a DataFrame where a boolean condition is met.
- Using the isin() method for filtering

```
In [57]: df.head() #display first 5 records
```

```
Out[57]:                    A         B         C         D
         2013-01-01 -0.287586 -2.112358  0.526925  0.882804
         2013-01-02  0.137791  0.786193 -1.043605  0.316305
         2013-01-03  0.538533 -0.059857 -0.212885 -0.005387
         2013-01-04 -0.896319  1.084214 -0.023529  1.932443
         2013-01-05 -0.144685 -0.741743  1.695067  0.689757
```

```
In [58]: df.tail(3) #display last 3 records
```

```
Out[58]:                    A         B         C         D
         2013-01-04 -0.896319  1.084214 -0.023529  1.932443
         2013-01-05 -0.144685 -0.741743  1.695067  0.689757
         2013-01-06  0.872990  0.804680  0.309702 -0.162149
```

```
In [59]: df.index #display indexes
```

```
Out[59]: DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
                        '2013-01-05', '2013-01-06'],
                       dtype='datetime64[ns]', freq='D')
```

```
In [60]: df.columns #display columns
```

```
Out[60]: Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
In [61]: df.values #print values
```

```
Out[61]: array([[-0.28758608, -2.11235816,  0.52692467,  0.88280383],
                [ 0.13779079,  0.78619285, -1.04360512,  0.31630524],
                [ 0.53853294, -0.05985702, -0.2128855 , -0.00538725],
                [-0.89631946,  1.08421387, -0.02352875,  1.93244302],
                [-0.144685  , -0.74174308,  1.69506731,  0.68975719],
                [ 0.8729897 ,  0.80468016,  0.30970165, -0.16214865]])
```

```
In [62]: #Transposung your data
         df.T
```

```
Out[62]:    2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
         A   -0.287586    0.137791    0.538533   -0.896319   -0.144685    0.872990
         B   -2.112358    0.786193   -0.059857    1.084214   -0.741743    0.804680
         C    0.526925   -1.043605   -0.212885   -0.023529    1.695067    0.309702
         D    0.882804    0.316305   -0.005387    1.932443    0.689757   -0.162149
```

```
In [63]: #Sorting by an axis
         df.sort_index(axis=1, ascending=False)
```

```
Out[63]:                    D         C         B         A
         2013-01-01  0.882804  0.526925 -2.112358 -0.287586
         2013-01-02  0.316305 -1.043605  0.786193  0.137791
         2013-01-03 -0.005387 -0.212885 -0.059857  0.538533
         2013-01-04  1.932443 -0.023529  1.084214 -0.896319
         2013-01-05  0.689757  1.695067 -0.741743 -0.144685
         2013-01-06 -0.162149  0.309702  0.804680  0.872990
```

```
In [64]: #Sorting by values
         df.sort_values(by='B')
```

```
Out[64]:                    A         B         C         D
         2013-01-01 -0.287586 -2.112358  0.526925  0.882804
         2013-01-05 -0.144685 -0.741743  1.695067  0.689757
         2013-01-03  0.538533 -0.059857 -0.212885 -0.005387
         2013-01-02  0.137791  0.786193 -1.043605  0.316305
         2013-01-06  0.872990  0.804680  0.309702 -0.162149
         2013-01-04 -0.896319  1.084214 -0.023529  1.932443
```

```
In [65]: #Describe shows a quick statistic summaryof your data
         df.describe()
```

```
Out[65]:                A         B         C         D
         count  6.000000  6.000000  6.000000  6.000000
         mean   0.036787 -0.039812  0.208612  0.608962
         std    0.627698  1.220446  0.907820  0.760369
         min   -0.896319 -2.112358 -1.043605 -0.162149
         25%   -0.251861 -0.571272 -0.165546  0.075036
         50%   -0.003447  0.363168  0.143086  0.503031
         75%    0.438347  0.800058  0.472619  0.834542
         max    0.872990  1.084214  1.695067  1.932443
```

```
In [66]: #Selecting a single column, which yields a Series, equivalent to af.
         df['A']

Out[66]: 2013-01-01   -0.287586
         2013-01-02    0.137791
         2013-01-03    0.538533
         2013-01-04   -0.896319
         2013-01-05   -0.144685
         2013-01-06    0.872990
         Freq: D, Name: A, dtype: float64

In [67]: #Selecting via[], which slices the rows.
         df[0:3]

Out[67]:                    A         B         C         D
         2013-01-01 -0.287586 -2.112358  0.526925  0.882804
         2013-01-02  0.137791  0.786193 -1.043605  0.316305
         2013-01-03  0.538533 -0.059857 -0.212885 -0.005387

In [68]: df['20130102':'20130104']

Out[68]:                    A         B         C         D
         2013-01-02  0.137791  0.786193 -1.043605  0.316305
         2013-01-03  0.538533 -0.059857 -0.212885 -0.005387
         2013-01-04 -0.896319  1.084214 -0.023529  1.932443

In [69]: #Selecting on a multi-axis by label
         df.loc[:,['A','B']]

Out[69]:                    A         B
         2013-01-01 -0.287586 -2.112358
         2013-01-02  0.137791  0.786193
         2013-01-03  0.538533 -0.059857
         2013-01-04 -0.896319  1.084214
         2013-01-05 -0.144685 -0.741743
         2013-01-06  0.872990  0.804680

In [70]: #Showing label slicing both endpoints are included
         df.loc['20130102':'20130104',['A','B']]

Out[70]:                    A         B
         2013-01-02  0.137791  0.786193
         2013-01-03  0.538533 -0.059857
         2013-01-04 -0.896319  1.084214

In [71]: #Reduction in the dimensions of the returned object
         df.loc['20130102',['A','B']]

Out[71]: A    0.137791
         B    0.786193
         Name: 2013-01-02 00:00:00, dtype: float64
```

```
In [72]: #For getting a scalar value
         df.loc[dates[0],'A']

Out[72]: -0.2875860803476811

In [73]: #For getting fast access to a scalar
         df.at[dates[0],'A']

Out[73]: -0.2875860803476811

In [74]: #Select via the position of the passed integers
         df.iloc[3]

Out[74]: A   -0.896319
         B    1.084214
         C   -0.023529
         D    1.932443
         Name: 2013-01-04 00:00:00, dtype: float64

In [75]: #By integer slices, acting similar to numpy/python
         df.iloc[3:5,0:2]

Out[75]:                    A         B
         2013-01-04 -0.896319  1.084214
         2013-01-05 -0.144685 -0.741743

In [76]: #By lists of integer position locations, similar to the numpy/python style
         df.iloc[[1,2,4],[0,2]]

Out[76]:                    A         C
         2013-01-02  0.137791 -1.043605
         2013-01-03  0.538533 -0.212885
         2013-01-05 -0.144685  1.695067

In [78]: #For slicing rows explicitly
         df.iloc[:,1:3]

Out[78]:                    B         C
         2013-01-01 -2.112358  0.526925
         2013-01-02  0.786193 -1.043605
         2013-01-03 -0.059857 -0.212885
         2013-01-04  1.084214 -0.023529
         2013-01-05 -0.741743  1.695067
         2013-01-06  0.804680  0.309702

In [79]: #For getting a value explicitly
         df.iloc[1,1]

Out[79]: 0.7861928459751845
```

```
In [80]: #Using a single columns value tto select data
         df[df.A > 0]
```

```
Out[80]:                    A         B         C         D
         2013-01-02  0.137791  0.786193 -1.043605  0.316305
         2013-01-03  0.538533 -0.059857 -0.212885 -0.005387
         2013-01-06  0.872990  0.804680  0.309702 -0.162149
```

```
In [81]: #Selecting values from a DataFrame where a boolean condition is met.
         df[df > 0]
```

```
Out[81]:                    A         B         C         D
         2013-01-01       NaN       NaN  0.526925  0.882804
         2013-01-02  0.137791  0.786193       NaN  0.316305
         2013-01-03  0.538533       NaN       NaN       NaN
         2013-01-04       NaN  1.084214       NaN  1.932443
         2013-01-05       NaN       NaN  1.695067  0.689757
         2013-01-06  0.872990  0.804680  0.309702       NaN
```

```
In [82]: #Using the isin() method for filtering:
         df2 = df.copy()
```

```
In [83]: df2['E'] = ['one','one','two','three','four','three']
         df2
```

```
Out[83]:                    A         B         C         D      E
         2013-01-01 -0.287586 -2.112358  0.526925  0.882804    one
         2013-01-02  0.137791  0.786193 -1.043605  0.316305    one
         2013-01-03  0.538533 -0.059857 -0.212885 -0.005387    two
         2013-01-04 -0.896319  1.084214 -0.023529  1.932443  three
         2013-01-05 -0.144685 -0.741743  1.695067  0.689757   four
         2013-01-06  0.872990  0.804680  0.309702 -0.162149  three
```

```
In [84]: df2[df2['E'].isin(['two','four'])]
```

```
Out[84]:                    A         B         C         D     E
         2013-01-03  0.538533 -0.059857 -0.212885 -0.005387   two
         2013-01-05 -0.144685 -0.741743  1.695067  0.689757  four
```

```
In [12]: import pandas as pd
         import csv


         with open('data.csv','r',newline='') as fileOpen:
             reader = csv.DictReader(fileOpen)
             list_of_data = []
             for line in reader:
                 list_of_data.append({'sepal length in cm':line['sepal length in cm'] , 'sepal
                                      'petal length in cm':line['petal length in cm'],'petal wi
```

```
                                'class':line['class']})
        #print(list_of_data)
        fileOpen.close()
        df = pd.DataFrame(list_of_data,
                          columns=['sepal length in cm','sepal width in cm','petal length in c
        df[20:100]
```

Out[12]:

	sepal length in cm	sepal width in cm	petal length in cm	petal width in cm	\
20	5.4	3.4	1.7	0.2	
21	5.1	3.7	1.5	0.4	
22	4.6	3.6	1.0	0.2	
23	5.1	3.3	1.7	0.5	
24	4.8	3.4	1.9	0.2	
25	5.0	3.0	1.6	0.2	
26	5.0	3.4	1.6	0.4	
27	5.2	3.5	1.5	0.2	
28	5.2	3.4	1.4	0.2	
29	4.7	3.2	1.6	0.2	
30	4.8	3.1	1.6	0.2	
31	5.4	3.4	1.5	0.4	
32	5.2	4.1	1.5	0.1	
33	5.5	4.2	1.4	0.2	
34	4.9	3.1	1.5	0.1	
35	5.0	3.2	1.2	0.2	
36	5.5	3.5	1.3	0.2	
37	4.9	3.1	1.5	0.1	
38	4.4	3.0	1.3	0.2	
39	5.1	3.4	1.5	0.2	
40	5.0	3.5	1.3	0.3	
41	4.5	2.3	1.3	0.3	
42	4.4	3.2	1.3	0.2	
43	5.0	3.5	1.6	0.6	
44	5.1	3.8	1.9	0.4	
45	4.8	3.0	1.4	0.3	
46	5.1	3.8	1.6	0.2	
47	4.6	3.2	1.4	0.2	
48	5.3	3.7	1.5	0.2	
49	5.0	3.3	1.4	0.2	
..	...	...	...	...	
70	5.9	3.2	4.8	1.8	
71	6.1	2.8	4.0	1.3	
72	6.3	2.5	4.9	1.5	
73	6.1	2.8	4.7	1.2	
74	6.4	2.9	4.3	1.3	
75	6.6	3.0	4.4	1.4	
76	6.8	2.8	4.8	1.4	
77	6.7	3.0	5.0	1.7	
78	6.0	2.9	4.5	1.5	

79	5.7	2.6	3.5	1.0
80	5.5	2.4	3.8	1.1
81	5.5	2.4	3.7	1.0
82	5.8	2.7	3.9	1.2
83	6.0	2.7	5.1	1.6
84	5.4	3.0	4.5	1.5
85	6.0	3.4	4.5	1.6
86	6.7	3.1	4.7	1.5
87	6.3	2.3	4.4	1.3
88	5.6	3.0	4.1	1.3
89	5.5	2.5	4.0	1.3
90	5.5	2.6	4.4	1.2
91	6.1	3.0	4.6	1.4
92	5.8	2.6	4.0	1.2
93	5.0	2.3	3.3	1.0
94	5.6	2.7	4.2	1.3
95	5.7	3.0	4.2	1.2
96	5.7	2.9	4.2	1.3
97	6.2	2.9	4.3	1.3
98	5.1	2.5	3.0	1.1
99	5.7	2.8	4.1	1.3

	class
20	Iris-setosa
21	Iris-setosa
22	Iris-setosa
23	Iris-setosa
24	Iris-setosa
25	Iris-setosa
26	Iris-setosa
27	Iris-setosa
28	Iris-setosa
29	Iris-setosa
30	Iris-setosa
31	Iris-setosa
32	Iris-setosa
33	Iris-setosa
34	Iris-setosa
35	Iris-setosa
36	Iris-setosa
37	Iris-setosa
38	Iris-setosa
39	Iris-setosa
40	Iris-setosa
41	Iris-setosa
42	Iris-setosa
43	Iris-setosa
44	Iris-setosa

```
45      Iris-setosa
46      Iris-setosa
47      Iris-setosa
48      Iris-setosa
49      Iris-setosa
..              ...
70  Iris-versicolor
71  Iris-versicolor
72  Iris-versicolor
73  Iris-versicolor
74  Iris-versicolor
75  Iris-versicolor
76  Iris-versicolor
77  Iris-versicolor
78  Iris-versicolor
79  Iris-versicolor
80  Iris-versicolor
81  Iris-versicolor
82  Iris-versicolor
83  Iris-versicolor
84  Iris-versicolor
85  Iris-versicolor
86  Iris-versicolor
87  Iris-versicolor
88  Iris-versicolor
89  Iris-versicolor
90  Iris-versicolor
91  Iris-versicolor
92  Iris-versicolor
93  Iris-versicolor
94  Iris-versicolor
95  Iris-versicolor
96  Iris-versicolor
97  Iris-versicolor
98  Iris-versicolor
99  Iris-versicolor

[80 rows x 5 columns]
```

```
In [18]: df.head()
```

```
Out[18]:   sepal length in cm sepal width in cm petal length in cm petal width in cm  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
```

```
            class
0  Iris-setosa
1  Iris-setosa
2  Iris-setosa
3  Iris-setosa
4  Iris-setosa
```

```
In [20]: df['class']
```

```
Out[20]: 0         Iris-setosa
         1         Iris-setosa
         2         Iris-setosa
         3         Iris-setosa
         4         Iris-setosa
         5         Iris-setosa
         6         Iris-setosa
         7         Iris-setosa
         8         Iris-setosa
         9         Iris-setosa
         10        Iris-setosa
         11        Iris-setosa
         12        Iris-setosa
         13        Iris-setosa
         14        Iris-setosa
         15        Iris-setosa
         16        Iris-setosa
         17        Iris-setosa
         18        Iris-setosa
         19        Iris-setosa
         20        Iris-setosa
         21        Iris-setosa
         22        Iris-setosa
         23        Iris-setosa
         24        Iris-setosa
         25        Iris-setosa
         26        Iris-setosa
         27        Iris-setosa
         28        Iris-setosa
         29        Iris-setosa
                     ...
         120    Iris-virginica
         121    Iris-virginica
         122    Iris-virginica
         123    Iris-virginica
         124    Iris-virginica
         125    Iris-virginica
         126    Iris-virginica
         127    Iris-virginica
```

```
128    Iris-virginica
129    Iris-virginica
130    Iris-virginica
131    Iris-virginica
132    Iris-virginica
133    Iris-virginica
134    Iris-virginica
135    Iris-virginica
136    Iris-virginica
137    Iris-virginica
138    Iris-virginica
139    Iris-virginica
140    Iris-virginica
141    Iris-virginica
142    Iris-virginica
143    Iris-virginica
144    Iris-virginica
145    Iris-virginica
146    Iris-virginica
147    Iris-virginica
148    Iris-virginica
149    Iris-virginica
Name: class, Length: 150, dtype: object
```

```
In [23]: df[98:100]
```

```
Out[23]:    sepal length in cm sepal width in cm petal length in cm petal width in cm  \
98                 5.1               2.5                3.0               1.1
99                 5.7               2.8                4.1               1.3

                 class
98  Iris-versicolor
99  Iris-versicolor
```

```
In [30]: new_df = df['sepal length in cm'] + df['sepal width in cm']
         new_df
```

```
Out[30]: 0      5.13.5
         1      4.93.0
         2      4.73.2
         3      4.63.1
         4      5.03.6
         5      5.43.9
         6      4.63.4
         7      5.03.4
         8      4.42.9
         9      4.93.1
         10     5.43.7
         11     4.83.4
```

```
12     4.83.0
13     4.33.0
14     5.84.0
15     5.74.4
16     5.43.9
17     5.13.5
18     5.73.8
19     5.13.8
20     5.43.4
21     5.13.7
22     4.63.6
23     5.13.3
24     4.83.4
25     5.03.0
26     5.03.4
27     5.23.5
28     5.23.4
29     4.73.2
        ...
120    6.93.2
121    5.62.8
122    7.72.8
123    6.32.7
124    6.73.3
125    7.23.2
126    6.22.8
127    6.13.0
128    6.42.8
129    7.23.0
130    7.42.8
131    7.93.8
132    6.42.8
133    6.32.8
134    6.12.6
135    7.73.0
136    6.33.4
137    6.43.1
138    6.03.0
139    6.93.1
140    6.73.1
141    6.93.1
142    5.82.7
143    6.83.2
144    6.73.3
145    6.73.0
146    6.32.5
147    6.53.0
148    6.23.4
```

```
149    5.93.0
Length: 150, dtype: object
```

```
In [33]: df[df['sepal length in cm'].isin(['5.0'])]
```

Out[33]:

	sepal length in cm	sepal width in cm	petal length in cm	petal width in cm	class
4	5.0	3.6	1.4	0.2	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
25	5.0	3.0	1.6	0.2	Iris-setosa
26	5.0	3.4	1.6	0.4	Iris-setosa
35	5.0	3.2	1.2	0.2	Iris-setosa
40	5.0	3.5	1.3	0.3	Iris-setosa
43	5.0	3.5	1.6	0.6	Iris-setosa
49	5.0	3.3	1.4	0.2	Iris-setosa
60	5.0	2.0	3.5	1.0	Iris-versicolor
93	5.0	2.3	3.3	1.0	Iris-versicolor