

**FYBCA-SEM2-204 - Programming Skills****UNIT-5: Python Collections and Library:****5.1 Python Collections:**

The collection Module in Python provides different types of containers. A Container is an object that is used to store different objects and provide a way to access the contained objects and iterate over them. Some of the built-in containers are Tuple, List, Dictionary, etc.

**5.1.1 Tuples : Declaring tuple, indexing tuple, changing tuple values, adding and removing data from tuple, Use of tuple() method to create tuple, count() and index() methods.**

**Tuples:** Tuples are used to store multiple items in a single variable. Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets → ()
- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

**Properties of Tuple:**

- 1. Ordered:** When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- 2. Unchangeable:** Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- 3. Allow Duplicates :** Since tuple are indexed, tuples can have items with the same value:

**➤ Creating a tuple**

A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets. The parentheses are optional but it is good practice to use. A tuple can be defined as follows.

```
T1 = (101, "Peter", 22)
T2 = ("Apple", "Banana", "Orange")
T3 = 10,20,30,40,50
print(type(T1))
print(type(T2))
print(type(T3))
```

**Output:**

```
<class 'tuple'>
<class 'tuple'>
<class 'tuple'>
```

A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

**Example of Different types of tuples**

```
my_tuple = ()                # Empty tuple
print(my_tuple)
my_tuple = (1, 2, 3)         # Tuple having integers
print(my_tuple)
my_tuple = (1, "Hello", 3.4) # tuple with mixed datatypes
print(my_tuple)
```

**FYBCA-SEM2-204 - Programming Skills**

```
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))      # nested tuple
print(my_tuple)
```

**Output:**

```
()
(1, 2, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
```

Creating a tuple with single element is slightly different. We will need to put comma after the element to declare the tuple.

```
tup1 = ("SDJIC")
print(type(tup1))
#Creating a tuple with single element
tup2 = ("SDJIC",)
print(type(tup2))
```

**Output:**

```
<class 'str'>
<class 'tuple'>
```

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value. Consider the following example of tuple:

```
tuple1 = (10, 20, 30, 40, 50, 60)
print(tuple1)
count = 0

for i in tuple1:
    print("tuple1[%d] = %d"%(count, i))
    count = count+1
```

**OUTPUT:**

```
(10, 20, 30, 40, 50, 60)
tuple1[0] = 10
tuple1[1] = 20
tuple1[2] = 30
tuple1[3] = 40
tuple1[4] = 50
tuple1[5] = 60
```

➤ **Tuple indexing and slicing**

The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to  $\text{length}(\text{tuple}) - 1$ . The items in the tuple can be accessed by using the index `[]` operator. Python also allows us to use the colon operator to access multiple items in the tuple.

**Consider the following example:**

```
tuple = (1,2,3,4,5,6,7)
print(tuple[1:])          #element 1 to end
print(tuple[:4])          #element 0 to 3 element
print(tuple[1:5])         #element 1 to 4 element
print(tuple[0:6:2])       #element 0 to 6 and take step of 2
```

**Output:**

```
(2, 3, 4, 5, 6, 7)
(1, 2, 3, 4)
(1, 2, 3, 4)
(1, 3, 5)
```



**FYBCA-SEM2-204 - Programming Skills**

Consider the following image to understand the indexing and slicing in detail.

Tuple = ( 0, 1, 2, 3, 4, 5 )					
0	1	2	3	4	5
Tuple[0] = 0		Tuple[0:] = (0, 1, 2, 3, 4, 5)			
Tuple[1] = 1		Tuple[:] = (0, 1, 2, 3, 4, 5)			
Tuple[2] = 2		Tuple[2:4] = (2, 3)			
Tuple[3] = 3		Tuple[1:3] = (1, 2)			
Tuple[4] = 4		Tuple[:4] = (0, 1, 2, 3)			
Tuple[5] = 5					

**Figure: Tuple indexing and slicing**

**Negative Indexing**

The tuple element can also access by using negative indexing. The index of -1 denotes the rightmost element and -2 to the second last item and so on.

The elements from left to right are traversed using the negative indexing. Consider the following example:

```
tuple1 = (1, 2, 3, 4, 5)
print(tuple1[-1])
print(tuple1[-4])
print(tuple1[-3:-1])
print(tuple1[: -1])
print(tuple1[-2:])
```

**Output:**

```
5
2
(3, 4)
(1, 2, 3, 4)
(4, 5)
```

**FYBCA-SEM2-204 - Programming Skills****Changing a Tuple**

Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like a list, its nested items can be changed. We can also assign a tuple to different values (reassignment).

```
# Changing tuple values
my_tuple = (4, 2, 3, [6, 5])
# my_tuple[1] = 9# TypeError: 'tuple' object does not support item assignment
my_tuple[3][0] = 9 # However, item of mutable element can be changed
print(my_tuple)
# Tuples can be reassigned
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple)
Output:
(4, 2, 3, [9, 5])
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

We can use **+** operator to combine two tuples. This is called **concatenation**. We can also **repeat** the elements in a tuple for a given number of times using the **\*** operator. Both **+** and **\*** operations result in a new tuple.

```
# Concatenation
print((1, 2, 3) + (4, 5, 6))

# Repeat
print(("Repeat",) * 3)
Output:
(1, 2, 3, 4, 5, 6)
('Repeat', 'Repeat', 'Repeat')
```

**Deleting a Tuple**

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple. Deleting a tuple entirely, however, is possible using the keyword **del**.

```
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
# can't delete items
# TypeError: 'tuple' object doesn't support item deletion
# del my_tuple[3]
# Can delete an entire tuple
del my_tuple
print(my_tuple) #NameError: name 'my_tuple' is not defined
Output:
Traceback (most recent call last):
  File "<string>", line 12, in <module>
NameError: name 'my_tuple' is not defined
```

**FYBCA-SEM2-204 - Programming Skills****Tuple Methods**

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Some examples of Python tuple methods:

```
my_tuple = ('a', 'p', 'p', 'l', 'e')
print(my_tuple.count('p')) # count the Occurrences of given element in tuple
print(my_tuple.index('l')) # return the index of element in tuple
```

**Output**

```
2
3
```

**The tuple() Constructor**

It is also possible to use the tuple() constructor to make a tuple.

**Example:Using the tuple() method to make a tuple:**

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

**Other Tuple Operations****1. Tuple Membership Test**

We can test if an item exists in a tuple or not, using the keyword in.

```
my_tuple = ('a', 'p', 'p', 'l', 'e') # Membership test in tuple
print('a' in my_tuple)
print('b' in my_tuple)
print('g' not in my_tuple)
```

**Output**

```
True
False
True
```

**2. Iterating Through a Tuple**

We can use a for loop to iterate through each item in a tuple.

```
# Using a for loop to iterate through a tuple
for name in ('John', 'Kate'):
    print("Hello", name)
```

**Output**

```
Hello John
Hello Kate
```

**Where use tuple?**

Using tuple instead of list is used in the following scenario.

1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.
2. Tuple can simulate a dictionary without keys. Consider the following nested structure, which can be used as a dictionary.

```
[(101, "John", 22), (102, "Mike", 28), (103, "Dustin", 30)]
```

**❖ List vs. Tuple**

Sr.	List	Tuple
1	The literal syntax of list is shown by the [].	The literal syntax of the tuple is shown by the ().
2	The List is mutable.	The tuple is immutable.
3	<b>Syntax:</b> list1 = [10, 'bhumika', 20]	<b>Syntax:</b> tup1 = (10, 'bhumika', 20)
4	The List has a variable length.	The tuple has the fixed length.
5	Implication of iterations is Time-consuming.	The implication of iterations is comparatively Faster.
6	The list is better for performing operations, such as insertion and deletion.	Tuple data type is appropriate for accessing the elements.
7	Lists consume more memory.	Tuple consume less memory as compared to the list.
8	Lists have several built-in methods.	Tuple does not have many built-in methods.
9	The unexpected changes and errors are more likely to occur.	In tuple, it is hard to take place.

**5.1.2 Sets: declaring set, access set data, set methods (add, clear, copy, discard, pop, remove, union, update).**

- A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).
- However, a set itself is mutable. We can add or remove items from it.
- Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

**Creating Python Sets**

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in **set()** function. It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# Different types of sets in Python
# set of integers
my_set = {1, 2, 3}
print(my_set)
```

```
# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

**Output**

```
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}
```

**Try the following examples as well.**

```
# set cannot have duplicates
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)
```

**FYBCA-SEM2-204 - Programming Skills**

```
my_set = set([1, 2, 3, 2])
print(my_set)
# set cannot have mutable items , here [3, 4] is a mutable list this will cause an error.
my_set = {1, 2, [3, 4]}
```

**Output:**

```
{1, 2, 3, 4}
{1, 2, 3}
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    my_set = {1, 2, [3, 4]}
TypeError: unhashable type: 'list'
```

*Creating an empty set is a bit tricky.*

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

```
# Distinguish set and dictionary while creating empty set
# initialize a with {}
a = {}
print(type(a))
# initialize a with set()
a = set()
# check data type of a
print(type(a))
```

**Output:**

```
<class 'dict'>
<class 'set'>
```

**Modifying a set in Python**

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

We can add a single element using the **add()** method, and multiple elements using the **update()** method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initialize my_set
my_set = {1, 3}
print(my_set)

my_set[0] # if you uncomment the above line you will get an error
# TypeError: 'set' object does not support indexing

my_set.add(2) # add an element
print(my_set) # Output: {1, 2, 3}

my_set.update([2, 3, 4]) # add multiple elements
print(my_set) # Output: {1, 2, 3, 4}
# add list and set
my_set.update([4, 5], {1, 6, 8}) # Output: {1, 2, 3, 4, 5, 6, 8}
print(my_set)
```

**FYBCA-SEM2-204 - Programming Skills****Removing elements from a set**

A particular item can be removed from a set using the methods **discard()** and **remove()**. The only difference between the two is that the **discard()** function leaves a set unchanged if the element is not present in the set. On the other hand, the **remove()** function will raise an error in such a condition (if element is not present in the set).

The following example will illustrate this.

```
# Difference between discard() and remove()
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print("Main set :",my_set)
my_set.discard(4)                # Output: {1, 3, 5, 6}
print("After Discard :", my_set)
my_set.remove(6)                 # Output: {1, 3, 5}
print("After Remove: ", my_set)
# discard an element not present in my_set
my_set.discard(2)                # Output: {1, 3, 5}
print("After Discard :",my_set)
# remove an element which is not present in my_set so that you will get an error.
my_set.remove(2)                 # Output: KeyError
```

- Similarly, we can remove and return an item using the **pop()** method.
- Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.
- We can also remove all the items from a set using the **clear()** method.

```
# initialize my_set
my_set = set("HelloWorld")        #Output: set of unique elements
print(my_set)
print("pop element is:", my_set.pop()) # Output: random element
my_set.pop()                     # pop another element
print("pop another element is:",my_set)
my_set.clear()                   # clear my_set
print(my_set)
Output:
{'H', 'W', 'o', 'e', 'l', 'r', 'd'}
pop element is: H
pop another element is: {'o', 'e', 'l', 'r', 'd'}
set()
```

**Python Set copy()**

The **copy()** method returns a shallow copy of the set.

A set can be copied using **=** operator in Python. For example:

```
numbers = {1, 2, 3, 4}
new_numbers = numbers
```

The problem with copying the set in this way is that if you modify the numbers set, the new\_numbers set is also modified.

```
numbers = {1, 2, 3, 4}
new_numbers = numbers
new_numbers.add(5)
print('numbers: ', numbers)
print('new_numbers: ', new_numbers)
```

**Output:**

```
numbers: {1, 2, 3, 4, 5}
new_numbers: {1, 2, 3, 4, 5}
```



**FYBCA-SEM2-204 - Programming Skills**

However, if you need the original set to be unchanged when the new set is modified, you can use the `copy()` method.

**The syntax of `copy()` is:**

```
set.copy()
```

**`copy()` Parameters**

It doesn't take any parameters.

**Return Value from `copy()`**

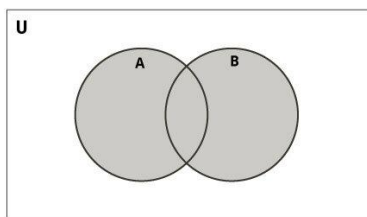
The `copy()` method returns a shallow copy of the set.

**Example 1: How the `copy()` method works for sets?**

```
numbers = {1, 2, 3, 4}
new_numbers = numbers.copy()
new_numbers.add(5)
print('numbers: ', numbers)
print('new_numbers: ', new_numbers)
```

**Output:**

```
numbers: {1, 2, 3, 4}
new_numbers: {1, 2, 3, 4, 5}
```

**Set Union in Python**


**Figure: Set Union in Python**

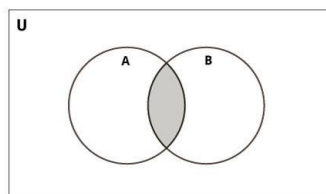
Union of A and B is a set of all elements from both sets.

Union is performed using `|` operator. Same can be accomplished using the **`union()`** method.

```
# Set union method # initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
print(A | B) # use | operator
print(A.union(B)) # use union function
print(B.union(A)) # use union function on B
```

**Output**

```
{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 4, 5, 6, 7, 8}
```

**Set Intersection in python**


**Figure: Set Intersection**

**FYBCA-SEM2-204 - Programming Skills**

Intersection of A and B is a set of elements that are common in both the sets.

Intersection is performed using & operator.

Same can be accomplished using the intersection() method.

```
# Intersection of sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
print(A & B)                #use & operator
print(A.intersection(B))
print(B.intersection(A))
```

**Output:**

```
{4, 5}
{4, 5}
{4, 5}
```

**5.1.3 Dictionary:**

Python Dictionary is used to store the data in a **key-value** pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the mutable data-structure. The dictionary is defined into element Keys and values.

- Keys must be a single element
- Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

**Ordered or Unordered?**

- As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.
- When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
- Unordered means that the item does not have a defined order, you cannot refer to an item by using an index.

**Changeable**

- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

**Duplicates Not Allowed**

- Dictionaries cannot have two items with the same key.

**5.1.3.1 Creating Dictionary, Adding, Accessing and Removing element****Creating the dictionary**

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).

**Syntax:**

```
Dict = {key1: value1, key2: value2}
```

**Example:**

```
Dict = {"Name": "Tom", "Age": 22}
```

In the above dictionary Dict, The keys Name and Age are the string that is an immutable object.

**Let's see an example to create a dictionary and print its content.**

```
Employee = {"Name": "John", "Age": 29, "salary": 25000, "Company": "GOOGLE"}
print(type(Employee))
```

**FYBCA-SEM2-204 - Programming Skills**

```
print("printing Employee data .... ")
print(Employee)
Output:
<class 'dict'>
Printing Employee data ....
{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
```

Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces **{}** is used to create empty dictionary.

```
Dict = {} # Creating an empty Dictionary
print("Empty Dictionary: ")
print(Dict)
# Creating a Dictionary with dict() method
Dict = dict({1: 'Java', 2: 'C', 3: 'PHP'})
print("\n Create Dictionary by using dict(): ")
print(Dict)
# Creating a Dictionary with each item as a Pair
Dict = dict([(1, 'Tony'), (2, 'Hulk')])
print("\n Dictionary with each item as a pair: ")
print(Dict)
Output:
Empty Dictionary:
{}
Create Dictionary by using dict():
{1: 'Java', 2: 'C', 3: 'PHP'}
Dictionary with each item as a pair:
{1: 'Tony', 2: 'Hulk'}
```

**Accessing the dictionary values**

We have discussed how the data can be accessed in the list and tuple by using the indexing. However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary. The dictionary values can be accessed in the following way.

```
Employee = {"Name": "MIHIR", "Age": 29, "salary": 25000, "Company": "GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print("Name : %s" %Employee["Name"])
print("Age : %d" %Employee["Age"])
print("Salary : %d" %Employee["salary"])
print("Company : %s" %Employee["Company"])
Output:
<class 'dict'>
printing Employee data ....
Name : MIHIR
Age : 29
Salary : 25000
Company : GOOGLE
```

Python provides us with an alternative to use the **get()** method to access the dictionary values. It would give the same result as given by the indexing.

**FYBCA-SEM2-204 - Programming Skills****Adding dictionary values**

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key Dict[key] = value. The **update()** method is also used to update an existing value.

**Note:** If the key-value already present in the dictionary, the value gets updated. Otherwise, the new keys added in the dictionary.

**Example - 1:** Let's see an example to update the dictionary values.

```
Dict = {} # Creating an empty Dictionary
print("Empty Dictionary: ")
print(Dict)
Dict[0] = 'MOHAN' # Adding elements to dictionary one at a time
Dict[2] = 'SOHAN'
Dict[3] = 'ROHAN'
print("\nDictionary after adding 3 elements: ")
print(Dict)
# Adding set of values with a single Key
# The Emp_ages doesn't exist to dictionary
Dict['Emp_ages'] = 20, 33, 24
print("\nDictionary after adding 3 elements: ")
print(Dict)
# Updating existing Key's Value
Dict[3] = 'NEERAJ'
print("\nUpdated key value: ")
print(Dict)
Dict.update({0: "White"})
print("Updated Dictionary : ", Dict)
```

**Output:**

Empty Dictionary:  
{}

Dictionary after adding 3 elements:  
{0: 'MOHAN', 2: 'SOHAN', 3: 'ROHAN'}

Dictionary after adding 3 elements:  
{0: 'MOHAN', 2: 'SOHAN', 3: 'ROHAN', 'Emp\_ages': (20, 33, 24)}

Updated key value:  
{0: 'MOHAN', 2: 'SOHAN', 3: 'NEERAJ', 'Emp\_ages': (20, 33, 24)}

Updated Dictionary: {0: 'White', 2: 'SOHAN', 3: 'NEERAJ', 'Emp\_ages': (20, 33, 24)}

**Example - 2:**

```
Employee = {"Name": "Sundar", "Age": 29, "salary": 25000, "Company": "GOOGLE"}
print("Type of Object:", type(Employee))
print("Printing Employee data .... ")
print(Employee)
print("Enter the details of the new employee....")
Employee["Name"] = input("Name: ")
Employee["Age"] = int(input("Age: "))
Employee["salary"] = int(input("Salary: "))
Employee["Company"] = input("Company: ")
```

**FYBCA-SEM2-204 - Programming Skills**

```
print("\nPrinting the new data...")
print(Employee)
Output:
<class 'dict'>
printing Employee data ....
{'Name': 'Sundar', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
Enter the details of the new employee....
Name: MOHAN
Age: 32
Salary: 36000
Company: MICROSOFT
printing the new data
{'Name': 'MOHAN', 'Age': 32, 'salary': 36000, 'Company': 'MICROSOFT'}
```

**Deleting or removing elements using del keyword**

The items of the dictionary can be deleted by using the **del** keyword as given below.

```
Employee = {"Name": "Bhumika", "Age": 29, "salary": 250000, "Company": "GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Deleting some of the employee data")
del Employee["Name"]
del Employee["Company"]
print("printing the modified information ")
print(Employee)
print("Deleting the dictionary: Employee")
del Employee
print("Lets try to print it again ")
print(Employee)
Output:
<class 'dict'>
printing Employee data ....
{'Name': 'Bhumika', 'Age': 29, 'salary': 250000, 'Company': 'GOOGLE'}
Deleting some of the employee data
printing the modified information
{'Age': 29, 'salary': 250000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined
```

**Note:** The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

**FYBCA-SEM2-204 - Programming Skills****5.1.3.2 Dictionary methods: get(), pop(), popitem(), clear(), copy()****1. Python Dictionary get()**

The get() method returns the value for the specified key if key is in dictionary.

**The syntax of get() is:**

```
dictionary.get(keyname, value)
```

**get() Parameters**

get() method takes maximum of two parameters:

- key - key to be searched in the dictionary
- value (optional) - Value to be returned if the key is not found. The default value is None.

**Return Value from get()**

get() method returns:

- the value for the specified key if key is in dictionary.
- None if the key is not found and value is not specified.
- value if the key is not found and value is specified.

**Example 1: How get() works for dictionaries?**

```
person = {'name': 'Phill', 'age': 22}
print('Name: ', person.get('name'))
print('Age: ', person.get('age'))
print('Salary: ', person.get('salary'))      # value is not provided
print('Salary: ', person.get('salary', 0.0))  # value is provided
```

**Output:**

```
Name: Phill
Age: 22
Salary: None
Salary: 0.0
```

**Python get() method Vs dict[key] to Access Elements**

get() method returns a default value if the key is missing.

However, if the key is not found when you use dict[key], KeyError exception is raised.

```
person = {}
print('Salary: ', person.get('salary'))      # Using get() results in None
print(person['salary'])                      # Using [] results in KeyError
```

**Output:**

```
Salary: None
Traceback (most recent call last):
  File "", line 7, in
    print(person['salary'])
KeyError: 'salary'
```

**2. Python Dictionary pop()**

The pop() method removes and returns an element from a dictionary having the given key.

**The syntax of pop() method is**

```
dictionary.pop(keyname, default)
```

**FYBCA-SEM2-204 - Programming Skills****pop() Parameters**

pop() method takes two parameters:

- key - key which is to be searched for removal
- default - value which is to be returned when the key is not in the dictionary

**Return value from pop()**

The pop() method returns:

- If key is found - removed/popped element from the dictionary
- If key is not found - value specified as the second argument (default)
- If key is not found and default argument is not specified - KeyError exception is raised

**Example 1: Pop an element from the dictionary**

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }
element = sales.pop('apple')
print('The popped element is:', element)
print('The dictionary is:', sales)
```

**Output:**

```
The popped element is: 2
The dictionary is: {'orange': 3, 'grapes': 4}
```

**Example 2: Pop an element not present from the dictionary**

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }
element = sales.pop('guava')
```

**Output:**

```
KeyError: 'guava'
```

**Example 3: Pop an element not present from the dictionary, provided a default value**

```
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }      # random sales dictionary
element = sales.pop('guava', 'banana')
print('The popped element is:', element)
print('The dictionary is:', sales)
```

**Output:**

```
The popped element is: banana
The dictionary is: {'orange': 3, 'apple': 2, 'grapes': 4}
```

**3. Python Dictionary popitem()**

The Python popitem() method removes and returns the last element (key, value) pair inserted into the dictionary.

**The syntax of popitem() is:**

```
dict.popitem()
```

Parameters for popitem() method

The popitem() doesn't take any parameters.

**Return Value from popitem() method**

The popitem() method removes and returns the (key, value) pair from the dictionary in the Last In, First Out (LIFO) order.

- Returns the latest inserted element (key,value) pair from the dictionary.
- Removes the returned element pair from the dictionary.

*Note: Before Python 3.7, the popitem() method returned and removed an arbitrary element (key, value) pair from the dictionary.*

**FYBCA-SEM2-204 - Programming Skills****Example: Working of popitem() method**

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0}
# ('salary', 3500.0) is inserted at the last, so it is removed.
result = person.popitem()
print('Return Value = ', result)
print('person = ', person)
person['profession'] = 'Plumber'      # inserting a new element pair
result = person.popitem()           # now ('profession', 'Plumber') is the latest element
print('Return Value = ', result)
print('person = ', person)
```

**Output:**

```
Return Value = ('salary', 3500.0)
person = {'name': 'Phill', 'age': 22}
Return Value = ('profession', 'Plumber')
person = {'name': 'Phill', 'age': 22}
```

**Note:** The popitem() method raises a KeyError error if the dictionary is empty.

**4. Python Dictionary clear()**

The clear() method removes all items from the dictionary. **The syntax of clear() is:**

```
dict.clear()
```

clear() Parameters : clear() method doesn't take any parameters.

**Return Value from clear()**

clear() method doesn't return any value (returns None).

**Example 1: How clear() method works for dictionaries?**

```
d = {1: "one", 2: "two"}
d.clear()
print('d =', d)
```

**Output:**

```
d = {}
```

**5. Python Dictionary copy()**

The copy() method returns a shallow copy of the dictionary.

**The syntax of copy() is:**

```
dict.copy()
```

**copy() Parameters**

copy() method doesn't take any parameters.

**Return Value from copy()**

This method returns a shallow copy of the dictionary. It doesn't modify the original dictionary.

**Example 1: How copy works for dictionaries?**

```
original = {1:'one', 2:'two'}
new = original.copy()
print('Original: ', original)
print('New: ', new)
```

**Output:**

```
Original: {1: 'one', 2: 'two'}
New: {1: 'one', 2: 'two'}
```

**Difference in Using copy() method, and = Operator to Copy Dictionaries**

When copy() method is used, a new dictionary is created which is filled with a copy of the references from the original dictionary.

When = operator is used, a new reference to the original dictionary is created.



**FYBCA-SEM2-204 - Programming Skills****Example 2: Using = Operator to Copy Dictionaries**

```
original = {1:'one', 2:'two'}  
new = original  
new.clear() # removing all elements from the list  
print('new: ', new)  
print('original: ', original)  
Output:  
new: {}  
original: {}
```

Here, when new dictionary is cleared, original dictionary is also cleared.

**Example 3: Using copy() to Copy Dictionaries**

```
original = {1:'one', 2:'two'}  
new = original.copy()  
new.clear() # removing all elements from the list  
print('new: ', new)  
print('original: ', original)  
Output:  
new: {}  
original: {1: 'one', 2: 'two'}
```

Here, when new dictionary is cleared, original dictionary remains unchanged.

**5.2 Introduction to Numpy and Pandas****5.2.1 Overview of numpy**

NumPy is a Python package. It stands for 'Numerical Python' for the computation and processing of the multidimensional and single dimensional array elements. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.

- **Travis Oliphant** created NumPy package in 2005 by injecting the features of the ancestor module Numeric into another module Numarray.
- It is an extension module of Python which is mostly written in C. It provides various functions which are capable of performing the numeric computations with a high speed.
- NumPy provides various powerful data structures, implementing multi-dimensional arrays and matrices. These data structures are used for the optimal computations regarding arrays and matrices.

**There are the following advantages of using NumPy for data analysis.**

1. NumPy performs array-oriented computing.
  2. It efficiently implements the multidimensional arrays.
  3. It performs scientific computations.
  4. It is capable of performing Fourier Transform and reshaping the data stored in multidimensional arrays.
  5. NumPy provides the in-built functions for linear algebra and random number generation.
- Nowadays, NumPy in combination with SciPy and Matplotlib is used as the replacement to MATLAB as Python is more complete and easier programming language than MATLAB.

**FYBCA-SEM2-204 - Programming Skills****Why is numpy faster than lists?**

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behavior is called locality of reference in computer science.
- This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

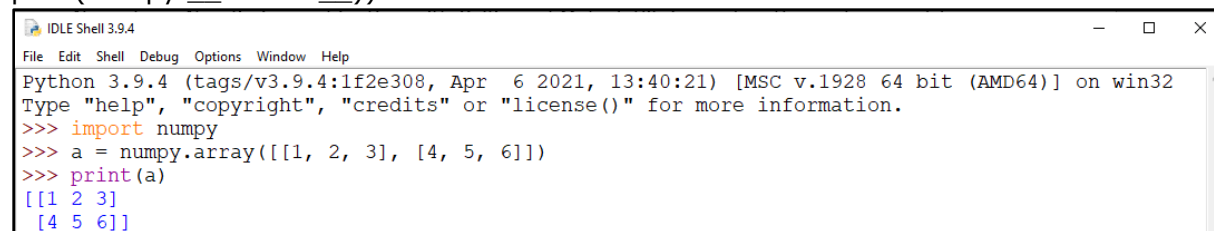
**How to Install numpy?**

NumPy doesn't come bundled with Python. We have to install it using the python pip installer. Execute the following command.

```
$ pip install numpy
```

It is best practice to install **NumPy** with the full SciPy stack. The binary distribution of the SciPy stack is specific to the operating systems. (pip stands for preferred installer program)

To verify the installation, open the Python prompt by executing python command on the terminal (cmd in the case of windows) and try to import the module NumPy as shown in the below image. If it doesn't give the error, then it is installed successfully. (Check version `print(numpy.__version__)`)



```
IDLE Shell 3.9.4
File Edit Shell Debug Options Window Help
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr 6 2021, 13:40:21) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import numpy
>>> a = numpy.array([[1, 2, 3], [4, 5, 6]])
>>> print(a)
[[1 2 3]
 [4 5 6]]
```

Alternatively, NumPy package is imported using the following syntax –

```
import numpy as np
```

NumPy is usually imported under the np alias.

**alias:** In Python alias are an alternate name for referring to the same thing.

Now the NumPy package can be referred to as **np** instead of numpy.

**NumPy ndarray**

- Ndarray is the n-dimensional array object defined in the numpy which stores the collection of the similar type of elements.
- In other words, we can define an ndarray as the collection of the data type (dtype) objects. The ndarray object can be accessed by using the 0 based indexing.
- Each element of the Array object contains the same size in the memory.
- In Numpy, number of dimensions of the array is called rank of the array. A tuple of integers giving the size of the array along each dimension is known as shape of the array.
- An array class in Numpy is called as ndarray. Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists.
- **Array** in Numpy is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.

**Creating a ndarray object**

NumPy is used to work with arrays. The array object in NumPy is called ndarray. We can create a NumPy ndarray object by using the `array()` function.

**FYBCA-SEM2-204 - Programming Skills**

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

**type():** This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

**Dimensions in Arrays**

A dimension in arrays is one level of array depth (nested arrays).

nested array: are arrays that have arrays as their elements.

**0-D Arrays**

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example: Create a 0-D array with value 35

```
import numpy as np
arr = np.array(35)
print(arr)
```

**1-D Arrays**

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

**Example: Create a 1-D array containing the values 1,2,3,4,5:**

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

**2-D Arrays**

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

**Example: Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:**

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

**Output:**

```
[[1 2 3]
 [4 5 6]]
```

**3-D arrays**

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

**Example: Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:**

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

**Check Number of Dimensions**

```
import numpy as np
```

**FYBCA-SEM2-204 - Programming Skills**

```
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

**Program for Basic array characteristics**

```
# Python program to demonstrate
# basic array characteristics
import numpy as np
# Creating array object
arr = np.array( [[ 1, 2, 3],
                 [ 4, 2, 5],
                 [ 7, 8, 6]] )

# Printing type of arr object
print("Array is of type: ", type(arr))
print("No. of dimensions: ", arr.ndim)    # Printing array dimensions (axes)
print("Shape of array: ", arr.shape)
print("Size of array: ", arr.size)        # Printing size (total number of elements) of array
print("Array stores elements of type: ", arr.dtype)
```

**OUTPUT:**

```
Array is of type: <class
'numpy.ndarray'>
No. of dimensions: 2
Shape of array: (3, 3)
Size of array: 9
Array stores elements of type: int32
```

**Some of the important attributes of a NumPy object are:**

**Ndim:** displays the dimension of the array

**Shape:** returns a tuple of integers indicating the size of the array

**Size:** returns the total number of elements in the NumPy array

**Dtype:** returns the type of elements in the array, i.e., int64, character

**Itemsize:** returns the size in bytes of each item

**Reshape:** Reshapes the NumPy array

**5.2.1.1 Numpy methods (Mean, Median, Mode, Standard Deviation and Variance)****5.2.1.2 Implementation of Numpy methods on numeric dataset created using list.**

Python is a very popular language when it comes to data analysis and statistics. In Machine Learning (and in mathematics) there are often three values that interest us:

Mean - The average value

Median - The midpoint value

Mode - The most common value

**Data Set:** In the mind of a computer, a data set is any collection of data. It can be anything from an array to a complete database.

Example of an array: [99,86,87,88,111,86,103,87,94,78,77,85,86]

**1. mean():** mean() function can be used to calculate mean/average of a given list of numbers. It returns mean of the data set passed as parameters.

Arithmetic mean is the sum of data divided by the number of data-points. It is a measure of the central location of data in a set of values which vary in range.

In Python, we usually do this by dividing the sum of given numbers with the count of number present.

**FYBCA-SEM2-204 - Programming Skills**

Given set of numbers: [n1, n2, n3, n4, n5, n6]

Sum of data-set = (n1 + n2 + n3 + n4 + n5)

Number of data produced = 5

Average or arithmetic mean = (n1 + n2 + n3 + n4 + n5) / 5

**Syntax :** mean([data-set])

**Parameters :**[data-set] : List or tuple of a set of numbers.

**Returns :** Sample arithmetic mean of the provided data-set.

**Exceptions :**TypeError when anything other than numeric values are passed as parameter.

**Python program to demonstrate mean() function**

```
import numpy as np
data1 = [1, 3, 4, 5, 7, 2]          # list of positive integer numbers
x = np.mean(data1)
print("Mean is :", x)              # Printing the mean
Output: Mean is : 3.6666666666666665
```

**2. median():** Compute the median of the given data (array elements) along the specified axis.

**How do we calculate median?**

Arrange them in ascending order

Median = middle term if total no. of terms are odd.

Median = Average of the terms in the middle (if total no. of terms are even)

**Syntax:**

numpy.median(arr, axis = None)

**Example1: Python program to demonstrate median() function**

```
import numpy as np
a1= [20, 2, 7, 1, 34]# 1D array
print("a1 : ", a1)
print("median of a1 : ", np.median(a1))
Output:
a1 : [20, 2, 7, 1, 34]
median of arr : 7.0
```

If there are two numbers in the middle, divide the sum of those numbers by two.

77, 78, 85, 86, 86, 86, 87, 87, 94, 98, 99, 103

(86 + 87) / 2 = 86.5

**Example2: Program to demonstrate median() function**

```
import numpy
arr = [99,86,87,88,86,103,87,94,78,77,85,86]
print("arr : ", arr)
x = numpy.median(arr)
print(x)
Output:
arr : [99, 86, 87, 88, 86, 103, 87, 94, 78, 77, 85, 86]
86.5
```

**FYBCA-SEM2-204 - Programming Skills**

**3. mode()** : Numpy does not have a method to find mode. So we use mode method from scipy (Scientific python) Package. It is used to compute the mode of the given data (array elements) along the specified axis. The Mode value is the value that appears the most number of times in data set.

i.e 99, 86, 87, 88, 111, 86, 103, 87, 94, 78, 77, 85, 86 = 86

**Example:**

```
#Use the SciPy mode() method to find the number that appears the most:
```

```
from scipy import stats
```

```
speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]
```

```
x = stats.mode(speed)
```

```
print(x)
```

**Output:**

```
ModeResult(mode=array([86]), count=array([3]))
```

The mode() method returns a ModeResult object that contains the mode number (86), and count (how many times the mode number appeared (3)).

[**Note:** ModuleNotFoundError: No module named 'scipy', when no scipy package is installed on computer. So install package before run mode())

```
$ pip install scipy (on cmd prompt of windows)]
```

**4. Standard Deviation**

**std()**:Compute the standard deviation of the given data (array elements) along the specified axis(if any). Standard Deviation (SD) is measured as the spread of data distribution in the given data set.

**Syntax:** numpy.std(arr, axis = None)

**Example:**numpy.std() method

```
import numpy as np
```

```
# 1D array
```

```
arr = [20, 2, 7, 1, 34]
```

```
print("arr : ", arr)
```

```
print("std of arr : ", np.std(arr))
```

```
OUTPUT:
```

```
arr : [20, 2, 7, 1, 34]
```

```
std of arr : 12.576167937809991
```

**5. Variance**

**var()**:Compute the Variance of the given data (array elements) along the specified axis(if any). Variance is another number that indicates how spread out the values is. In fact, if you take the square root of the variance, you get the standard deviation!Or the other way around, if you multiply the standard deviation by itself, you get the variance!

**Syntax:** numpy.var(arr, axis = None)

**Example:** NumPy var() method to find the variance:

```
import numpy
```

```
speed = [32,111,138,28,59,77,97]
```

```
x = numpy.var(speed)
```

```
print(x)
```

**FYBCA-SEM2-204 - Programming Skills****5.2.2 Pandas Dataframe**

- A panda is defined as an open-source library that provides high-performance data manipulation in Python.
- The name of Pandas is derived from the word **Panel Data**, which means an **Econometrics from Multidimensional data**.
- It is used for data analysis in Python and developed by **Wes McKinney** in 2008.
- Data analysis requires lots of processing, such as restructuring, cleaning or merging, etc. There are different tools are available for fast data processing, such as **Numpy, Scipy, Cython, and Panda**. But we prefer Pandas because working with Pandas is fast, simple and more expressive than other tools.
- Pandas is built on top of the Numpy package, means Numpy is required for operating the Pandas.

**What is Pandas?**

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.

**Why Use Pandas?**

Pandas allow us to analyze big data and make conclusions based on statistical theories. Pandas can clean messy data sets, and make them readable and relevant. Relevant data is very important in data science.

**What Can Pandas Do?**

Pandas give you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?

Pandas are also able to delete rows that are not relevant, or contain wrong values, like empty or NULL values. This is called cleaning the data.

**Installation of Pandas**

If you have Python and PIP already installed on a system, then installation of Pandas is very easy. Install it using this command:

```
$pip install pandas
```

**Import Pandas**

Once Pandas is installed, import it in your applications by adding the import keyword:

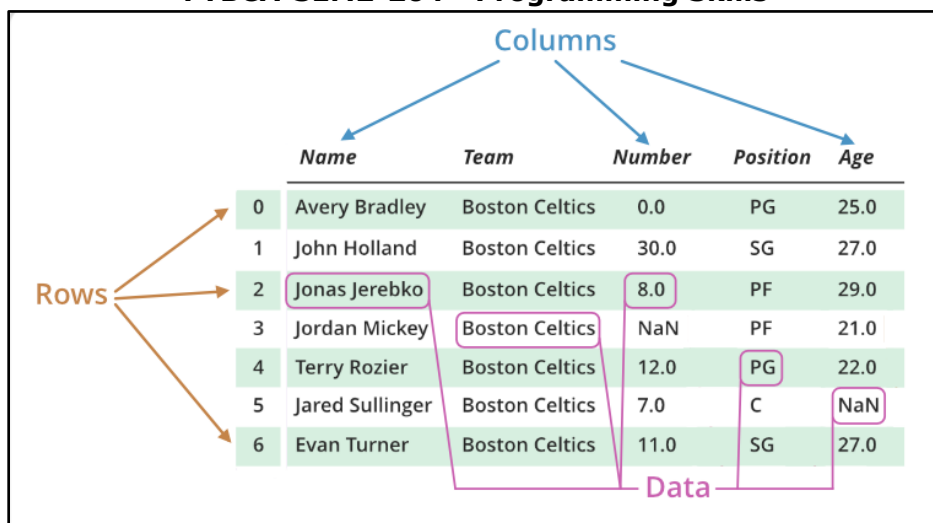
```
import pandas
```

OR

```
import pandas as pd
```

**5.2.2.1 Creating dataframe using list****Pandas DataFrame**

- Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).
- A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.
- Pandas DataFrame consists of three principal components, the data, rows, and columns.

**FYBCA-SEM2-204 - Programming Skills**


	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

**Figure: Pandas DataFrame**

In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, and Excel file. Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionary etc. Dataframe can be created in different ways here are some ways by which we create a dataframe.

**pandas.DataFrame**

A pandas DataFrame can be created using the following constructor –

`pandas.DataFrame( data, index, columns, dtype, copy)`

The parameters of the constructor are as follows –

Sr.No	Parameter & Description
1	<b>data:</b> data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.
2	<b>index:</b> For the row labels, the Index to be used for the resulting frame is Optional Default <code>np.arange(n)</code> if no index is passed.
3	<b>columns:</b> For column labels, the optional default syntax is - <code>np.arange(n)</code> . This is only true if no index is passed.
4	<b>dtype:</b> Data type of each column.
5	<b>copy:</b> This command (or whatever it is) is used for copying of data, if the default is False.

**Create DataFrame:** A pandas DataFrame can be created using various inputs like –

1. Lists
2. dict
3. Series
4. Numpy ndarrays
5. Another DataFrame

**Create an Empty DataFrame**

A basic DataFrame which can be created is an Empty Dataframe.

**Example: (emptydf.py)**

```
#import the pandas library and aliasing as pd
import pandas as pd
df = pd.DataFrame()
print(df)
```

**Output:**  
Empty DataFrame  
Columns: []  
Index: []



**FYBCA-SEM2-204 - Programming Skills**

**Creating a DataFrame using list:** DataFrame can be created using single list or list of lists. **Example1: The DataFrame can be created using a single list.**

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print(df)
```

**OUTPUT:**

```
0
0 1
1 2
2 3
3 4
4 5
```

**Example2: The DataFrame can be created using a list of lists.(Filename: listdf.py)**

```
import pandas as pd
lst = [['Vruti', 20, 89], ['Binjal',19, 85],
      ['Mihir', 19,90], ['Taher', 18,95],['Dhaval', 20,84]]
#Calling DataFrame constructor on list
df = pd.DataFrame(lst,columns=['Name','Age','Percentile'],dtype=float)
print(df)
```

**OUTPUT:**

	Name	Age	Percentile
0	Vruti	20.0	89.0
1	Binjal	19.0	85.0
2	Mihir	19.0	90.0
3	Taher	18.0	95.0
4	Dhaval	20.0	84.0

**5.2.2.2 Creating dataframe using dict of equal length list**

To create DataFrame from dict of narray/list, all the narray must be of same length. If index is passed then the length index should be equal to the length of arrays. If no index is passed, then by default, index will be range(n) where n is the array length.

**Example1: (FileName: dict\_df.py)**

```
import pandas as pd
data = {'Name':['Prince', 'Bhumika', 'Jalak', 'Jemil'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print(df)
```

**OUTPUT:**

	Name	Age
0	Prince	28
1	Bhumika	34
2	Jalak	29
3	Jemil	42

**Note:** Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n)

**FYBCA-SEM2-204 - Programming Skills**
**Example2: Let us now create indexed DataFrame using arrays. (FileName: dict\_df.py)**

```
import pandas as pd
data = {'Name':['Umang', 'Vanshita', 'Pratham', 'Saumil'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print(df)
```

**OUTPUT:**

	Name	Age
rank1	Umang	28
rank2	Vanshita	34
rank3	Pratham	29
rank4	Saumil	42

**Note:** Observe, the index parameter assigns an index to each row.

**5.2.2.3 Reading data using csv file (read\_csv())**
**Read CSV Files**

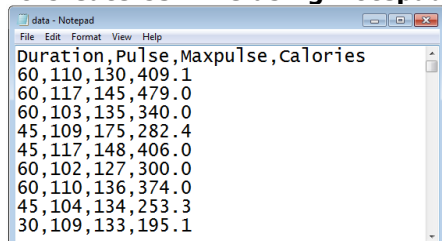
A simple way to store big data sets is to use CSV files (comma separated files). CSV files contain plain text and are a well know format that can be read by everyone including Pandas. With CSV files all you need is a single line to load in the data. In our examples we will be using a CSV file called 'data.csv'.

**Example: Load the CSV into a DataFrame(FileName: readcsv.py)**

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.to_string())
#print(df)
```

**Tip:** use to\_string() to print the entire DataFrame. By default, when you print a DataFrame, you will only get the first 5 rows, and the last 5 rows:

**To Create CSV file using Notepad or Excel. (FileName: data.csv)**


**5.2.2.4 Retrieving rows and columns from dataframe using index**
**5.2.2.5 Retrieving rows and columns using loc and iloc functions.**
**Indexing in Pandas:**

Indexing in pandas means simply selecting particular rows and columns of data from a DataFrame. Indexing could mean selecting all the rows and some of the columns, some of the rows and all of the columns, or some of each of the rows and columns. Indexing can also be known as Subset Selection.

Selection could be:

- Selecting some rows and some columns
- Selecting some rows and all columns
- Selecting all rows and some columns

**FYBCA-SEM2-204 - Programming Skills**

**Let's create a simple dataframe with a list of tuples, say column names are: 'Name', 'Age', 'City' and 'Salary'.**

```
import pandas as pd
#List of Tuples
employees = [('Stuti', 28, 'Varanasi', 20000),
             ('Saumya', 32, 'Delhi', 25000),
             ('Aditya', 25, 'Mumbai', 40000),
             ('Rohan', 32, 'Delhi', 35000),
             ('Ram', 32, 'Delhi', 30000),
             ('Salman', 32, 'Mumbai', 20000),
             ('Pooja', 40, 'Dehradun', 24000),
             ('Seema', 32, 'Delhi', 70000)]
# Create a DataFrame object from list
df = pd.DataFrame(employees, columns=['Name', 'Age', 'City', 'Salary'])
# Show the dataframe
print(df)
```

**Method 1: using Dataframe [ ]**

Indexing operator is used to refer to the square brackets [ ] following an object. [ ] is used to select a column by mentioning the respective column name.

**Example1: To select single column & to select multiple columns (indxoper.py)**

```
import pandas as pd          # import pandas
# List of Tuples
employees = [('Stuti', 28, 'Varanasi', 20000),
             ('Saumya', 32, 'Delhi', 25000),
             ('Aditya', 25, 'Mumbai', 40000),
             ('Rohan', 32, 'Delhi', 35000),
             ('Ram', 32, 'Delhi', 30000),
             ('Salman', 32, 'Mumbai', 20000),
             ('Pooja', 40, 'Dehradun', 24000),
             ('Seema', 32, 'Delhi', 70000)]

# Create a DataFrame object from list
df = pd.DataFrame(employees, columns=['Name', 'Age', 'City', 'Salary'])

print("\n<-----Show the dataframe----->")
print(df)                                #Show the dataframe

print("\n<-----Result Of Cities----->")
result = df["City"]                      #Using the operator [ ] to select a column
print(result)

# Using the operator [ ] to select multiple columns
print("\n<-----Result Of Name, Age & Salary----->")
result = df[["Name", "Age", "Salary"]]
print(result)
```

**FYBCA-SEM2-204 - Programming Skills**
**Output:**

```
<-----Show the dataframe----->
   Name  Age  City  Salary
0  Stuti   28  Varanasi  20000
1  Saumya  32   Delhi  25000
2  Aditya  25   Mumbai  40000
3   Rohan  32   Delhi  35000
4    Ram   32   Delhi  30000
5  Salman  32   Mumbai  20000
6  Pooja  40  Dehradun  24000
7   Seema  32   Delhi  70000

<-----Result of Cities----->
0  Varanasi
1    Delhi
2   Mumbai
3    Delhi
4    Delhi
5   Mumbai
6  Dehradun
7    Delhi
Name: City, dtype: object

<-----Result of Name, Age & Salaries----->
   Name  Age  Salary
0  Stuti   28  20000
1  Saumya  32  25000
2  Aditya  25  40000
3   Rohan  32  35000
4    Ram   32  30000
5  Salman  32  20000
6  Pooja  40  24000
7   Seema  32  70000
```

**Method 2: Using DataFrame.loc[ ]**

.loc[ ] the function selects the data by labels of rows or columns. It can select a subset of rows and columns. There are many ways to use this function.

**Syntax:**

```
DataFrame.loc["row1"]
```

Pandas **set\_index()** is a method to set a List, Series or Data frame as index of a Data Frame. Index column can be set while making a data frame too. But sometimes a data frame is made out of two or more data frames and hence later index can be changed using this method.

**Syntax:**

```
DataFrame.set_index(keys, drop=True, append=False, inplace=False, verify_integrity=False)
```

**Parameters:**

**keys:** Column name or list of column name.

**drop:** Boolean value which drops the column used for index if True.

**append:** Appends the column to existing index column if True.

**inplace:** Makes the changes in the dataframe if True.

**verify\_integrity:** Checks the new index column for duplicates if True.

**Example1: To select single row & multiple rows.**

```
import pandas as pd
employees = [('Stuti', 28, 'Varanasi', 20000),
             ('Saumya', 32, 'Delhi', 25000),
             ('Aditya', 25, 'Mumbai', 40000),
             ('Rohan', 32, 'Delhi', 35000),
             ('Ram', 32, 'Delhi', 30000),
             ('Salman', 32, 'Mumbai', 20000),
             ('Pooja', 40, 'Dehradun', 24000),
             ('Seema', 32, 'Delhi', 70000)]
```

**FYBCA-SEM2-204 - Programming Skills**

```
# Create a DataFrame object from list
df = pd.DataFrame(employees,columns=['Name', 'Age', 'City', 'Salary'])
print("<-----DataFrame----->")
print(df)

# Set 'Name' column as index on a Dataframe
print("\n<-----DataFrame after Set_index----->")
df.set_index("Name", inplace = True)
print(df)

# Using the operator .loc[] to select single row
result = df.loc["Stuti"]

print("\n<----DataFrame With Single Row----->")
print(result)

# Using the operator .loc[] to select multiple rows
result = df.loc[["Stuti", "Seema"]]
print("\n<----DataFrame With multiple rows----->")
print(result)
```

**Example2: To select multiple rows and particular columns.****Syntax:**

```
Dataframe.loc[["row1", "row2"...], ["column1", "column2", "column3"...]]
```

```
# To select multiple rows and particular columns.
import pandas as pd
# List of Tuples
employees = [('Stuti', 28, 'Varanasi', 20000),
             ('Saumya', 32, 'Delhi', 25000),
             ('Aditya', 25, 'Mumbai', 40000),
             ('Rohan', 32, 'Delhi', 35000),
             ('Ram', 32, 'Delhi', 30000),
             ('Salman', 32, 'Mumbai', 20000),
             ('Pooja', 40, 'Dehradun', 24000),
             ('Seema', 32, 'Delhi', 70000)]

# Create a DataFrame object from list
df = pd.DataFrame(employees,columns=['Name', 'Age','City', 'Salary'])

# Set 'Name' column as index on a Dataframe
df.set_index("Name", inplace = True)

# Using the operator .loc[] to select multiple rows with some particular columns
result = df.loc[["Stuti", "Seema"],["City", "Salary"]]

print(result)                                # Show the dataframe
```

**Method 3: Using Dataframe.iloc[ ]**

iloc[ ] is used for selection based on position. It is similar to loc[] indexer but it takes only integer values to make selections.

**FYBCA-SEM2-204 - Programming Skills**

**Example1: to select a single row, multiple rows, multiple rows with some particular columns and to select all the rows with some particular columns**

```
# import pandas
import pandas as pd
employees = [('Stuti', 28, 'Varanasi', 20000),
             ('Saumya', 32, 'Delhi', 25000),
             ('Aditya', 25, 'Mumbai', 40000),
             ('Rohan', 32, 'Delhi', 35000),
             ('Ram', 32, 'Delhi', 30000),
             ('Salman', 32, 'Mumbai', 20000),
             ('Pooja', 40, 'Dehradun', 24000),
             ('Seema', 32, 'Delhi', 70000)]

# Create a DataFrame object from list
df = pd.DataFrame(employees, columns=['Name', 'Age', 'City', 'Salary'])
print("<-----DataFrame----->")
print(df)

result = df.iloc[2]                                # Using the operator .iloc[] to select single row
print("\n<----single row----->")                # Show the dataframe
print(result)

# Using the operator .iloc[] to select multiple rows
result = df.iloc[[2, 3, 5]]
print("\n<----multiple rows----->")            # Show the dataframe
print(result)

# to select multiple rows with some particular columns.
# Using the operator .iloc[] to select multiple rows with some particular columns
result = df.iloc[[2, 3, 5], [0, 1]]
print("\n<--multiple rows with some particular columns-->")
print(result)

# Using the operator .iloc[] to select all the rows with some particular columns
result = df.iloc[:, [0, 1]]
print("\n<---select all the rows with some particular columns-->")
print(result)
```

**Output:**

```
<-----DataFrame----->
   Name  Age  City  Salary
0  Stuti   28  Varanasi  20000
1  Saumya  32   Delhi  25000
2  Aditya  25   Mumbai  40000
3   Rohan  32   Delhi  35000
4    Ram   32   Delhi  30000
5  Salman  32   Mumbai  20000
6   Pooja  40  Dehradun  24000
7   Seema  32   Delhi  70000

<----single row----->
Name    Aditya
Age         25
City    Mumbai
Salary    40000
Name: 2, dtype: object

<----multiple rows----->
   Name  Age  City  Salary
2  Aditya  25  Mumbai  40000
3   Rohan  32   Delhi  35000
5  Salman  32  Mumbai  20000

<--multiple rows with some particular columns-->
   Name  Age
2  Aditya  25
3   Rohan  32
5  Salman  32

<---select all the rows with some particular columns-->
   Name  Age
0  Stuti   28
1  Saumya  32
2  Aditya  25
3   Rohan  32
4    Ram   32
5  Salman  32
6   Pooja  40
7   Seema  32
```