Module-3

Maheshkumar N

Module-3

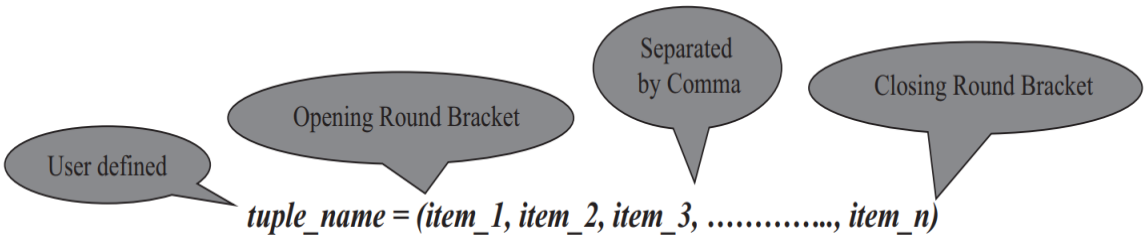
- **Dictionaries:** Creating Dictionary, Accessing and Modifying key: value Pairs in Dictionaries, Built-In Functions Used on Dictionaries, Dictionary Methods, The del Statement,
- **Tuples and Sets:** Creating Tuples, Basic Tuple Operations, Indexing and Slicing in Tuples, Built-In Functions Used on Tuples, Relation between Tuples and Lists, Relation between Tuples and Dictionaries, Tuple Methods, Sets, Set Methods, Traversing of Sets, Frozenset.

Tuples

- In mathematics, a tuple is a finite ordered list (sequence) of elements.
- A tuple is defined as a data structure that comprises an ordered, finite sequence of immutable, heterogeneous elements that are of fixed sizes.
- Often, you may want to return more than one value from a function. Tuples solve this problem.
- A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

Creating Tuples

- A tuple is a finite ordered list of values of possibly different types which is used to bundle related values together without having to create a specific type to hold them.
- Tuples are immutable. Once a tuple is created, you cannot change its values.
- A tuple is defined by putting a comma-separated list of values inside parentheses ( ).
- Each value inside a tuple is called an item.
- The syntax for creating tuples is,




```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Creating Tuples

- You can access tuple items by referring to the index number, inside square brackets:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

- What is the output of this ?

```
thistuple = ("apple",)
print(type(thistuple))


thistuple = ("apple")
print(type(thistuple))
```

Creating Tuples

- You can create an empty tuple without any values. The syntax is,

tuple_name = ()

```
1  empty_tuple = ()
2  print(empty_tuple)
3  print(type(empty_tuple))
```

```
()
<class 'tuple'>
```

Creating Tuples

- You can store any item of type string, number, object, another variable, and even another tuple itself. You can have a mix of different types of items in tuples, and they need not be homogeneous.

```
air_force = ("f15", "f22a", "f35a")
fighter_jets = (1988, 2005, 2016, air_force)
print(fighter_jets)
print(air_force)
```

Basic Tuple Operations

- Like in lists, you can use the + operator to concatenate tuples together and the * operator to repeat a sequence of tuple items.

```
1  tuple_1 = (2, 0, 1, 4)
2  tuple_2 = (2, 0, 1, 9)
3  print(tuple_1 + tuple_2)
4  print(tuple_1 * 3)
5  print(tuple_1 == tuple_2)
```

```
(2, 0, 1, 4, 2, 0, 1, 9)
(2, 0, 1, 4, 2, 0, 1, 4, 2, 0, 1, 4)
False
```

- ***in*** and ***not in*** membership operators

```
1  tuple_items = (1, 9, 8, 8)
2  print(1 in tuple_items)
3  print(25 in tuple_items)
```

```
True
False
```

Basic Tuple Operations

- Comparison operators like <, <=, >, >=, == and != are used to compare tuples. For example

```
1  tuple_1 = (9, 8, 7)
2  tuple_2 = (9, 1, 1)
3  print(tuple_1 > tuple_2)
4  print(tuple_1 != tuple_2)
```

```
True
True
```

The tuple() Function

- The built-in tuple() function is used to create a tuple. The syntax for the tuple() function is,

tuple([sequence])

- where the sequence can be a number, string or tuple itself.
- If the optional sequence is not specified, then an empty tuple is created.

```
norse = "vikings"
string_to_tuple = tuple(norse)
print(string_to_tuple)
zeus = ["g", "o", "d", "o", "f", "s", "k", "y"]
list_to_tuple = tuple(zeus)
print(list_to_tuple)
print(string_to_tuple + tuple("scandinavia"))
print(list_to_tuple + tuple(["g", "r", "e", "e", "k"]))
letters = ("a", "b", "c")
numbers = (1, 2, 3)
nested_tuples = (letters, numbers)
print(nested_tuples)
print(tuple("wolverine"))
```

```
('v', 'i', 'k', 'i', 'n', 'g', 's')
('g', 'o', 'd', 'o', 'f', 's', 'k', 'y')
('v', 'i', 'k', 'i', 'n', 'g', 's', 's', 'c', 'a', 'n', 'd', 'i', 'n', 'a', 'v', 'i', 'a')
('g', 'o', 'd', 'o', 'f', 's', 'k', 'y', 'g', 'r', 'e', 'e', 'k')
(('a', 'b', 'c'), (1, 2, 3))
('w', 'o', 'l', 'v', 'e', 'r', 'i', 'n', 'e')
```

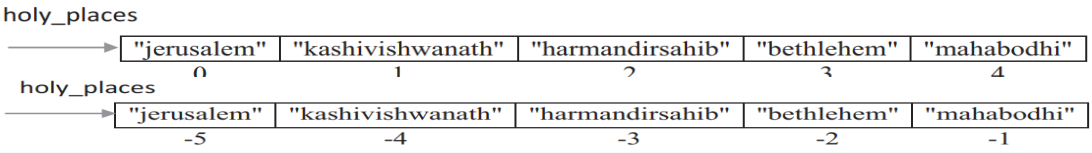
Indexing and Slicing in Tuples

- Each item in a tuple can be called individually through indexing.
- The expression inside the bracket is called the index.
- Square brackets [ ] are used by tuples to access individual items, with the first item at index 0, the second item at index 1 and so on.
- The index provided within the square brackets indicates the value being accessed.
- The syntax for accessing an item in a tuple is,

tuple_name[index]

where index should always be an integer value and indicates the item to be selected

Indexing and Slicing in Tuples




```python
holy_places = ("jerusalem", "kashivishwanath", "harmandirsahib", "bethlehem", "mahabodhi")
print(holy_places)
print(holy_places[0])
print(holy_places[1])
print(holy_places[2])
print(holy_places[3])
print(holy_places[4])
print(holy_places[-2])
print(holy_places[5])
```

```
('jerusalem', 'kashivishwanath', 'harmandirsahib', 'bethlehem', 'mahabodhi')
jerusalem
kashivishwanath
harmandirsahib
bethlehem
mahabodhi
bethlehem

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-17-97a6fed773ab> in <module>
      7 print(holy_places[4])
      8 print(holy_places[-2])
----> 9 print(holy_places[5])

IndexError: tuple index out of range
```

Slicing in Tuples

- Slicing of tuples is allowed in Python wherein a part of the tuple can be extracted by specifying an index range along with the colon (:) operator, which itself results as tuple type.
- The syntax for tuple slicing is,

  tuple_name[start:stop[:step]]

- where both start and stop are integer values (positive or negative values).
- Tuple slicing returns a part of the tuple from the start index value to stop index value, which includes the start index value but excludes the stop index value.
- The step specifies the increment value to slice by and it is optional.

Slicing in Tuples

colors →

"v"	"i"	"b"	"g"	"y"	"o"	"r"
0	1	2	3	4	5	6
-7	-6	-5	-4	-3	-2	-1

```
1  colors = ("v", "i", "b", "g", "y", "o", "r")
2  print(colors)
3  print(colors[1:4])
4  print(colors[:5])
5  print(colors[3:])
6  print(colors[:])
7  print(colors[::])
8  print(colors[1:5:2])
9  print(colors[::2])
10 print(colors[::-1])
11 print(colors[-5:-2])
```

```
('v', 'i', 'b', 'g', 'y', 'o', 'r')
('i', 'b', 'g')
('v', 'i', 'b', 'g', 'y')
('g', 'y', 'o', 'r')
('v', 'i', 'b', 'g', 'y', 'o', 'r')
('v', 'i', 'b', 'g', 'y', 'o', 'r')
('i', 'g')
('v', 'b', 'y', 'r')
('r', 'o', 'y', 'g', 'b', 'i', 'v')
('b', 'g', 'y')
```

Built-In Functions Used on Tuples

Built-In Functions Used on Tuples

Built-In Functions	Description
len()	The *len()* function returns the numbers of items in a tuple.
sum()	The *sum()* function returns the sum of numbers in the tuple.
sorted()	The *sorted()* function returns a sorted copy of the tuple as a list while leaving the original tuple untouched.

```
1  years = (1987, 1985, 1981, 1996)
2  print(len(years))
3  print(sum(years))
4  sorted_years = sorted(years)
5  print(sorted_years)
```

```
4
7949
[1981, 1985, 1987, 1996]
```

Relation between Tuples and Lists

- Though tuples may seem similar to lists, they are often used in different situations and for different purposes.
- Tuples are immutable, and usually, contain a heterogeneous sequence of elements that are accessed via unpacking or indexing.
- Lists are mutable, and their items are accessed via indexing.
- Items cannot be added, removed or replaced in a tuple.

```
1 coral_reef = ("great_barrier", "ningaloo_coast", "amazon_reef", "pickles_reef")
2 coral_reef[0] = "pickles_reef"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-33d925c9c166> in <module>
      1 coral_reef = ("great_barrier", "ningaloo_coast", "amazon_reef", "pickles_reef")
----> 2 coral_reef[0] = "pickles_reef"

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

Relation between Tuples and Lists

- Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

```
1 coral_reef = ("great_barrier", "ningaloo_coast", "amazon_reef", "pickles_reef")
2 coral_reef_list = list(coral_reef)
3 print(coral_reef_list)
```

```
['great_barrier', 'ningaloo_coast', 'amazon_reef', 'pickles_reef']
```

- The items in a tuple is mutable, then you can change it. By converting tuple into list

Relation between Tuples and Lists

- If an item within a tuple is mutable, then you can change it.
- Consider the presence of a list as an item in a tuple, then any changes to the list get reflected on the overall items in the tuple.

```
1 german_cars = ["porsche", "audi", "bmw"]
2 european_cars = ("ferrari", "volvo", "renault", german_cars)
3 print(european_cars)
4 european_cars[3].append("mercedes")
5 print(german_cars)
6 print(european_cars)
```

```
('ferrari', 'volvo', 'renault', ['porsche', 'audi', 'bmw'])
['porsche', 'audi', 'bmw', 'mercedes']
('ferrari', 'volvo', 'renault', ['porsche', 'audi', 'bmw', 'mercedes'])
```

Tuple Methods

Various Tuple Methods

Tuple Methods	Syntax	Description
count()	tuple_name.count(item)	The *count()* method counts the number of times the item has occurred in the tuple and returns it.
index()	tuple_name.index(item)	The *index()* method searches for the given item from the start of the tuple and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the tuple, then *ValueError* is thrown by this method.

Note: Replace the word *"tuple_name"* mentioned in the syntax with your *actual tuple name* in your code.

```
1 channels = ("ngc", "discovery", "animal_planet", "history", "ngc")
2 print(channels.count("ngc"))
3 print(channels.index("history"))
```

```
2
3
```

Tuple Packing and Unpacking

- The statement t = 12345, 54321, 'hello!' is an example of tuple packing.

```
1 t = 12345, 54321, 'hello!'
2 print(t)
```

```
(12345, 54321, 'hello!')
```

- The reverse operation of tuple packing is also possible. This operation is called tuple unpacking and works for any sequence on the right-hand side.

```
1 x, y, z = t
2 print(x)
3 print(y)
4 print(z)
```

```
12345
54321
hello!
```

Program

- Program to Iterate Over Items in Tuples Using for Loop

```
1 ocean_animals = ("electric_eel", "jelly_fish", "shrimp", "turtles", "blue_whale")
2 def main():
3     for each_animal in ocean_animals:
4         print(f"{each_animal} is an ocean animal")
5 if __name__ == "__main__":
6     main()
```

```
electric_eel is an ocean animal
jelly_fish is an ocean animal
shrimp is an ocean animal
turtles is an ocean animal
blue_whale is an ocean animal
```

Populating tuples with items

```python
tuple_items = ()
total_items = int(input("Enter the total number of items: "))
for i in range(total_items):
    user_input = int(input("Enter a number: "))
    tuple_items += (user_input,)
print(f"Items added to tuple are {tuple_items}")
list_items = []
total_items = int(input("Enter the total number of items: "))
for i in range(total_items):
    item = input("Enter an item to add: ")
    list_items.append(item)
items_of_tuple = tuple(list_items)
print(f"Tuple items are {items_of_tuple}")
```

```
Enter the total number of items: 4
Enter a number: 4
Enter a number: 3
Enter a number: 2
Enter a number: 1
Items added to tuple are (4, 3, 2, 1)
Enter the total number of items: 4
Enter an item to add: 1
Enter an item to add: 2
Enter an item to add: 3
Enter an item to add: 4
Tuple items are ('1', '2', '3', '4')
```

Dictionaries

- Creating Dictionary
- Accessing and Modifying key : value Pairs in Dictionaries
- Built-In Functions Used on Dictionaries
- Dictionary Methods
- The del Statement,

Dictionaries

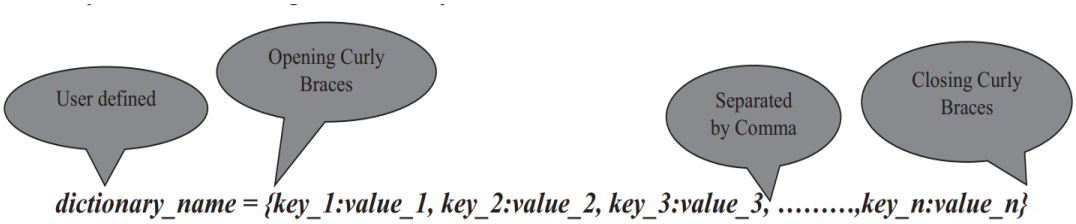
- A dictionary is a collection of an unordered set of key:value pairs, with the requirement that the keys are unique within a dictionary.
- Dictionaries are constructed using curly braces { }, wherein you include a list of key:value pairs separated by commas.
- Also, there is a colon (:) separating each of these key and value pairs, where the words to the left of the colon operator are the keys and the words to the right of the colon operator are the values.
- Unlike lists, which are indexed by a range of numbers, dictionaries are indexed by keys.
- Here a key along with its associated value is called a ***key:value*** pair.
- Dictionary keys are case sensitive.

Dictionaries

- Dictionary keys are immutable type and can be either a string or a number.
- Since lists can be modified in place using index assignments, slice assignments, or methods like *append()* and *extend(),* you cannot use lists as keys.
- Duplicate keys are not allowed in the dictionary.

Dictionaries

- The syntax for creating a dictionary is




```
fish = {"g": "goldfish", "s":"shark", "n": "needlefish", "b":"barramundi", "m":"mackerel"}
print(fish)
```

```
{'g': 'goldfish', 's': 'shark', 'n': 'needlefish', 'b': 'barramundi', 'm': 'mackerel'}
```

The way the dictionaries are entered and the dictionaries are displayed in the output

Dictionaries

- Dictionaries and their associated values:

```
mixed_dict = {"portable":"laptop", 9:11, 7:"julius"}
print(mixed_dict)
print(type(mixed_dict))
```

```
{'portable': 'laptop', 9: 11, 7: 'julius'}
<class 'dict'>
```

- Creating empty dictionaries:

```
empty_dictionary = {}
print(empty_dictionary)
print(type(empty_dictionary))
```

```
{}
<class 'dict'>
```

Dictionaries

- Accessing and Modifying key : value Pairs in Dictionaries:
- The syntax for accessing the value for a key in the dictionary is,

dictionary_name[key]

- The syntax for modifying the value of an existing key or for adding a new key:value pair to a dictionary is,

dictionary_name[key] = value

Dictionaries

- Adding and Modifying key:value pairs:
- If the key is already present in the dictionary, then the key gets updated with the new value.
- If the key is not present then the new key:value pair gets added to the dictionary.

```
renaissance = {"giotto":1305, "donatello":1440, "michelangelo":1511, "botticelli":1480, "clouet":1520}
renaissance["giotto"] = 1310
print(renaissance)
print(renaissance["michelangelo"])
renaissance["leonardo"] = 1503
print(renaissance)
print(renaissance["piero"])
```

```
{'giotto': 1310, 'donatello': 1440, 'michelangelo': 1511, 'botticelli': 1480, 'clouet': 1520}
1511
{'giotto': 1310, 'donatello': 1440, 'michelangelo': 1511, 'botticelli': 1480, 'clouet': 1520, 'leonardo': 1503}

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-4-30b6be945717> in <module>
      5 renaissance["leonardo"] = 1503
      6 print(renaissance)
----> 7 print(renaissance["piero"])

KeyError: 'piero'
```

Dictionaries

- Check for the presence of a key in the dictionary:
- You can check for the presence of a key in the dictionary using in and not in membership operators.
- It returns either a Boolean True or False value.

```
clothes = {"rainy":"raincoats", "summer":"tees", "winter":"sweaters"}
print("spring" in clothes)
print("spring" not in clothes)
```

```
False
True
```

Dictionaries

- The dict() Function:
- The built-in ***dict()*** function is used to create dictionary.
- The syntax for ***dict()*** function when the optional keyword arguments used is,

dict([**kwarg])

```
numbers = dict(one=1, two=2, three=3)
print(numbers)
```

```
{'one': 1, 'two': 2, 'three': 3}
```

- The syntax for ***dict()*** function when iterables used is,

dict(iterable[, **kwarg])

- You can specify an iterable containing exactly two objects as tuple, the key and value in the ***dict()*** function.

```
print(dict([('sape', 4139), ('guido', 4127), ('jack', 4098)]))
```

```
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dictionaries

- Built-in functions used on dictionaries:

Built-In Functions Used on Dictionaries

Built-in Functions	Description
len()	The *len()* function returns the number of items (*key:value* pairs) in a dictionary.
all()	The *all()* function returns Boolean True value if all the keys in the dictionary are True else returns False.
any()	The *any()* function returns Boolean True value if any of the key in the dictionary is True else returns False.
sorted()	The *sorted()* function by default returns a list of items, which are sorted based on dictionary keys.

- In Python, any non-zero integer value is True, and zero is interpreted as False.
- The sorted() function returns the sorted list of keys by default in ascending order without modifying the original ***key:value*** pairs

Dictionaries

- Built-in functions used on dictionaries:

```
presidents = {"washington":1732, "jefferson":1751, "lincoln":1809, "roosevelt":1858, "eisenhower":1890}
print(len(presidents))
all_dict_func = {0:True, 2:False}
print(all(all_dict_func))
all_dict_func = {1:True, 2:False}
print(all(all_dict_func))
any_dict_func = {1:True, 2:False}
print(any(any_dict_func))
print(sorted(presidents))
print(sorted(presidents, reverse = True))
print(sorted(presidents.values()))
print(sorted(presidents.items()))
```

```
5
False
True
True
['eisenhower', 'jefferson', 'lincoln', 'roosevelt', 'washington']
['washington', 'roosevelt', 'lincoln', 'jefferson', 'eisenhower']
[1732, 1751, 1809, 1858, 1890]
[('eisenhower', 1890), ('jefferson', 1751), ('lincoln', 1809), ('roosevelt', 1858), ('washington', 1732)]
```

Dictionary Methods:

clear()	dictionary_name.clear()	The *clear()* method removes all the *key:value* pairs from the dictionary.
fromkeys()	dictionary_name.fromkeys(seq [, value])	The *fromkeys()* method creates a new dictionary from the given sequence of elements with a value provided by the user.
get()	dictionary_name.get(key [, default])	The *get()* method returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to *None*, so that this method never raises a *KeyError*.
items()	dictionary_name.items()	The *items()* method returns a new view of dictionary's key and value pairs as tuples.
keys()	dictionary_name.keys()	The *keys()* method returns a new view consisting of all the keys in the dictionary.
pop()	dictionary_name.pop(key[, default])	The *pop()* method removes the key from the dictionary and returns its value. If the key is not present, then it returns the default value. If default is not given and the key is not in the dictionary, then it results in *KeyError*.
popitem()	dictionary_name.popitem()	The *popitem()* method removes and returns an arbitrary (key, value) tuple pair from the dictionary. If the dictionary is empty, then calling *popitem()* results in *KeyError*.
setdefault()	dictionary_name.setdefault (key[, default])	The *setdefault()* method returns a value for the key present in the dictionary. If the key is not present, then insert the key into the dictionary with a default value and return the default value. If key is present, *default* defaults to None, so that this method never raises a *KeyError*.
update()	dictionary_name.update([other])	The *update()* method updates the dictionary with the *key:value* pairs from *other* dictionary object and it returns *None*.
values()	dictionary_name.values()	The *values()* method returns a new view consisting of all the values in the dictionary.

Dictionary Methods:

```python
box_office_billion = {
    "avatar": 2009,
    "titanic": 1997,
    "starwars": 2015,
    "harrypotter": 2011,
    "avengers": 2012,
}
box_office_billion_fromkeys = box_office_billion.fromkeys(box_office_billion)
print(box_office_billion_fromkeys)
box_office_billion_fromkeys = box_office_billion.fromkeys(
    box_office_billion, "billion_dollar"
)
print(box_office_billion_fromkeys)
print(box_office_billion.get("frozen"))
print(box_office_billion.get("frozen", 2013))
print(box_office_billion.keys())
print(box_office_billion.values())
print(box_office_billion.items())
box_office_billion.update({"frozen": 2013})
print(box_office_billion)
box_office_billion.setdefault("minions")
print(box_office_billion)
box_office_billion.setdefault("ironman", 2013)
print(box_office_billion)
print(box_office_billion.pop("avatar"))
print(box_office_billion.popitem())
box_office_billion.clear()
print(box_office_billion)
```

Dictionary Methods:

```
{'avatar': None, 'titanic': None, 'starwars': None, 'harrypotter': None, 'avengers': None}
{'avatar': 'billion_dollar', 'titanic': 'billion_dollar', 'starwars': 'billion_dollar', 'harrypotter': 'billion_dollar', 'aveng
ers': 'billion_dollar'}
None
2013
dict_keys(['avatar', 'titanic', 'starwars', 'harrypotter', 'avengers'])
dict_values([2009, 1997, 2015, 2011, 2012])
dict_items([('avatar', 2009), ('titanic', 1997), ('starwars', 2015), ('harrypotter', 2011), ('avengers', 2012)])
{'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012, 'frozen': 2013}
{'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012, 'frozen': 2013, 'minions': None}
{'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011, 'avengers': 2012, 'frozen': 2013, 'minions': None, 'ir
onman': 2013}
2009
('ironman', 2013)
{}
```

Dictionary Methods:

- The del() Statement:
- To delete the key:value pair, use the del statement followed by the name of the dictionary along with the key you want to delete.

del dict_name[key]

```
1 animals = {"r":"raccoon", "c":"cougar", "m":"moose"}
2 print(animals)
3 del animals["c"]
4 print(animals)
```

```
{'r': 'raccoon', 'c': 'cougar', 'm': 'moose'}
{'r': 'raccoon', 'm': 'moose'}
```

Program to Dynamically Build User Input as a List :

```python
def main():
    print("Method 1: Building Dictionaries")
    build_dictionary = {}
    for i in range(0, 2):
        dic_key = input("Enter key ")
        dic_val = input("Enter val ")
        build_dictionary.update({dic_key: dic_val})
    print(f"Dictionary is {build_dictionary}")

    print("Method 2: Building Dictionaries")
    build_dictionary = {}
    for i in range(0, 2):
        dic_key = input("Enter key ")
        dic_val = input("Enter val ")
        build_dictionary[dic_key] = dic_val
    print(f"Dictionary is {build_dictionary}")

    print("Method 3: Building Dictionaries")
    build_dictionary = {}
    i = 0
    while i < 2:
        dict_key = input("Enter key ")
        dict_val = input("Enter val ")
        build_dictionary.update({dict_key: dict_val})
        i = i + 1
    print(f"Dictionary is {build_dictionary}")


if __name__ == "__main__":
    main()
```

```
Method 1: Building Dictionaries
Enter key microsoft
Enter val windows
Enter key canonical
Enter val ubuntu
Dictionary is {'microsoft': 'windows', 'canonical': 'ubuntu'}
Method 2: Building Dictionaries
Enter key apple
Enter val macos
Enter key canonical
Enter val ubuntu
Dictionary is {'apple': 'macos', 'canonical': 'ubuntu'}
Method 3: Building Dictionaries
Enter key microsoft
Enter val windows
Enter key apple
Enter val macos
Dictionary is {'microsoft': 'windows', 'apple': 'macos'}
```

Program to Illustrate Traversing of key:value Pairs in Dictionaries Using for Loop:

```python
currency = {
    "India": "Rupee",
    "USA": "Dollar",
    "Russia": "Ruble",
    "Japan": "Yen",
    "Germany": "Euro",
}


def main():
    print("List of Countries")
    for key in currency.keys():
        print(key)
    print("List of Currencies in different Countries")
    for value in currency.values():
        print(value)
    for key, value in currency.items():
        print(f"'{key}' has a currency of type '{value}'")


if __name__ == "__main__":
    main()
```

```
List of Countries
India
USA
Russia
Japan
Germany
List of Currencies in different Countries
Rupee
Dollar
Ruble
Yen
Euro
'India' has a currency of type 'Rupee'
'USA' has a currency of type 'Dollar'
'Russia' has a currency of type 'Ruble'
'Japan' has a currency of type 'Yen'
'Germany' has a currency of type 'Euro'
```

Write Python Program to Check for the Presence of a Ke... in the Dictionary and to Sum All Its Values:

```python
historical_events = {
    "apollo11": 1969,
    "great_depression": 1929,
    "american_revolution": 1775,
    "berlin_wall": 1989,
}


def check_key_presence():
    key = input("Enter the key to check for its presence ")
    if key in historical_events.keys():
        print(f"Key '{key}' is present in the dictionary")
    else:
        print(f"Key '{key}' is not present in the dictionary")


def sum_dictionary_values():
    print("Sum of all the values in the dictionary is")
    print(f"{sum(historical_events.values())}")


def main():
    check_key_presence()
    sum_dictionary_values()


if __name__ == "__main__":
    main()
```

```
Enter the key to check for its presence apollo11
Key 'apollo11' is present in the dictionary
Sum of all the values in the dictionary is
7662
```

Write Python Program to Count the Number of Times a Item Appears in the List:

```python
novels = ["gone_girl", "davinci_code", "games_of_thrones", "gone_girl", "davinci_code"]


def main():
    count_items = dict()
    for book_name in novels:
        count_items[book_name] = count_items.get(book_name, 0) + 1
    print("Number of times a novel appears in the list is")
    print(count_items)


if __name__ == "__main__":
    main()
```

```
Number of times a novel appears in the list is
{'gone_girl': 2, 'davinci_code': 2, 'games_of_thrones': 1}
```

Write Python Program to Count the Number of Times Each Word Appears in a Sentence

```python
def main():
    count_words = dict()
    sentence = input("Enter a sentence ")
    words = sentence.split()
    for each_word in words:
        count_words[each_word] = count_words.get(each_word, 0) + 1
    print("The number of times each word appears in a sentence is")
    print(count_words)


if __name__ == "__main__":
    main()
```

```
Enter a sentence Everyone needs a little inspiration from time to time
The number of times each word appears in a sentence is
{'Everyone': 1, 'needs': 1, 'a': 1, 'little': 1, 'inspiration': 1, 'from': 1, 'time': 2, 'to': 1}
```

Write Python Program to Generate a Dictionary That Contains (i: i*i) Such that i Is a Number Ranging from 1 to n

```python
def main():
    number = int(input("Enter a number "))
    create_number_dict = dict()
    for i in range(1, number + 1):
        create_number_dict[i] = i * i
    print("The generated dictionary of the form (i: i*i) is")
    print(create_number_dict)


if __name__ == "__main__":
    main()
```

```
Enter a number 3
The generated dictionary of the form (i: i*i) is
{1: 1, 2: 4, 3: 9}
```

Write a Program That Accepts a Sentence and Calculate the Number of Digits, Uppercase and Lowercase Letters.

```python
def main():
    number = int(input("Enter a number "))
    create_number_dict = dict()
    for i in range(1, number + 1):
        create_number_dict[i] = i * i
    print("The generated dictionary of the form (i: i*i) is")
    print(create_number_dict)


if __name__ == "__main__":
    main()
```

```
Enter a number 3
The generated dictionary of the form (i: i*i) is
{1: 1, 2: 4, 3: 9}
```

Program to Demonstrate Nested Dictionaries

```python
student_details = {
    "name": "jasmine",
    "registration_number": "1AIT18CS05",
    "sub_marks": {"python": 95, "java": 90, ".net": 85},
}


def nested_dictionary():
    print(f"Student Name {student_details['name']}")
    print(f"Registration Number {student_details['registration_number']}")
    average = sum(student_details["sub_marks"].values()) / len(
        student_details["sub_marks"]
    )
    print(f"Average of all the subjects is {average}")


def main():
    nested_dictionary()


if __name__ == "__main__":
    main()
```

```
Student Name jasmine
Registration Number 1AIT18CS05
Average of all the subjects is 90.0
```