1. Introduction

A variable associates a name with a value, making it easy to remember and use the value later. Variables can store data of different types, and different types can do different things.

A data type consists of a set of values and a set of operations that can be performed on those values. Python has the following built-in data types.

Text Type: **str**
Numeric Types: **int**, **float**, **complex**
Sequence Types: **list**, **tuple**, **range**
Mapping Type: **dict**
Set Types: **set**, **frozenset**
Boolean Type: **bool**
Binary Types: **bytes**, **bytearray**, **memoryview**

The **type()** function is used to get the data type of any object and data type is set when you assign a value to a variable. For example,

```
x = 5
print(type(x))
```

If you want to specify the data type, you can use the constructor function. For example,

```
x = int(5)
```

Remember: Constructor function is also used for type casting. For example,

```
x = 56.78
print(int(x))
```

2. Numeric Types

- Integer (int) – Integer is a whole number, positive or negative, without decimals, of unlimited length. For example
  a = 567
- Floating Point Number (float) – a number, positive or negative, containing one or more decimals. It can also be scientific numbers with an "e" to indicate the power of 10. For example,
  x = 346.78
  y = 3.5e3
- Complex numbers (complex) – are written with a "j" as the imaginary part. For example,
  z = 2 + 3j

Remember: We can convert one type to another with **int()**, **float()**, and **complex()** methods. We cannot convert complex numbers into another number.

3. Strings

A string is a sequence of zero or more characters. Strings in python are surrounded by either single quotation marks, or double quotation marks. For example,

```
a = "Hello"
print(a)
```

Multiline strings are surrounded by three double quotes or single quotes. For example,

"""The best things
in life are free!"""

- **Strings and Arrays:** Strings are arrays of bytes representing Unicode characters. Python does not have a character data type and a single character is simply a string with length 1. Square brackets can be used to access elements of the string. The first character in the string has the position 0.

  a = "Hello, World!"
  print(a[1])

  We can loop through the characters in a string, with a for loop. For example,

  a = "Hello, World!"
  for x in a:
  print(x)

  To get the length of a string, use the **len()** function. For example,

  a = "Hello, World!"
  print(len(a))

  To check if a certain phrase or character is present in a string, we can use the keyword **in** (opposite is **not in**). For example,

  txt = "The best things in life are free!"
  print("free" in txt)

- **Slicing Strings:** We can slice a range of characters by specifying start index and end index separated by a colon. For example,

  str = "Master's in Data Science"
  print(str[3:7])

  The range will start at the first character if we leave out the start index. For example,

  print(str[:7])

  The range will go to the end if we leave out the end index. For example,

  print(str[2:])

  Use negative indexes to start the slice from the end of the string. For example,

  print(str[-5: -2])

  Remember: We can also mix both positive and negative indices while slicing strings.

- **Modifying Strings:** Python has a set of built-in methods to modify strings. The ***upper()*** method returns the string in upper case and ***lower()*** method returns the string in lower case. For example,

  str = "Master's in Data Science "
  print(str.upper())
  print(str.lower())

  The ***strip()*** method removes any whitespace from the beginning or the end of the string. For example,

  srt.strip()

The ***replace()*** method replaces a string with another string. For example,

```
str = "Master's in Data Science "
print(str.replace("Data", "Information"))
```

The ***split()*** method returns a list where the text between the specified separator becomes the list items.

```
str = "Master's in Information Technology"
print(str.split(""))
```

- **Concatenating Strings:** Two strings are concatenated using + operator. For example,

```
str1 = "Kathmandu "
str2 = "Nepal"
print(str1 + str2)
```

- **Converting Numbers to Strings:** Constructor function **str()** is used to convert numbers to strings. For example,

```
str1 = "Nepal, "
str2 = str(977)
print(str1 + str2)
```

To convert strings to numbers, we also use constructor function. For example, to convert string to integer we use **int()**.

```
a = 42
b = int("56")
print(a + b)
```

- **String Formatting:** The ***format()*** method formats the specified value(s) and insert them inside the string's placeholder. The placeholder is defined using curly brackets: {}. The *format()* method returns the formatted string. This method takes unlimited number of arguments, and are placed into the respective placeholders. For example

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

The placeholders can be identified using numbered indexes. Also, if you want to refer to the same value more than once, use the index number. For example,

```
quantity = 3
itemno = 567
price = 49
myorder = """I want {0} pieces of item number {1}
and price for item number {1} is {2:.2f} dollars."""
print(myorder.format(quantity, itemno, price))
```

You can also use named indexes by entering a name inside the curly bracket. For example,

```
myorder = """I want {quantity} pieces of item number {itemno}
```

and price for item number {itemno} is {price:.2f} dollars."""
print(myorder.format(quantity = 3, itemno = 567, price = 49))

Inside the placeholders you can add a formatting type to format the result. The different formatting types are:

:<	Left aligns the result (within the available space)
:>	Right aligns the result (within the available space)
:^	Center aligns the result (within the available space)
:=	Places sign to the leftmost position
:+	Use a plus sign to indicate if the result is positive or negative
:-	Use a minus sign for negative values only
:	Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers)
:,	Use a comma as a thousand separator
:_	Use a underscore as a thousand separator
:b	Binary format
:c	Converts the value into the corresponding unicode character
:d	Decimal format
:e	Scientific format, with a lower case e
:E	Scientific format, with a upper case E
:f	Fix point number format
:F	Fix point number format, in uppercase format (show inf and nan as INF and NAN)
:g	General format
:G	General format (using a upper case E for scientific notations)
:o	Octal format
:x	Hex format, lower case
:X	Hex format, upper case
:n	Number format
:%	Percentage format

For example,

quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:<8.2f} dollars."
print(myorder.format(quantity, itemno, price))

Remember: We can also use leading f character preceding the string literal to format strings. The idea behind f-String in Python is to make string interpolation simpler. To create an f-string, prefix the string with the letter f as given in the example below.

quantity = 3
itemno = 567
price = 49
myorder = f'I want {quantity} pieces of item number {itemno} for {price:.2f} dollars.'
print(myorder)

- **Escape Sequences:** Escape sequences are the way Python expresses special characters, such as the tab, the newline, and the backspace (delete key), as

literals. These are also used to insert characters that are illegal in a string. Backslash (\) is used for escape sequences. Some common escape sequences:

\a	Bell. For example, print("\a")
\n	New line. For example, print("Hello\nhi")
\t	Tab. For example, For example, print("Hello\thi")
\b	Backspace. For example, print("Hello \bhi")
\r	Carriage return. For example, print("Hello\rHi")
\'	Single quote. For example, print('It\'s alright.')
\"	Double quote. For example, print("I love \"Nepal\"")
\\	Backslash. For example, print("Hello\\Hi")
\ooo	Octal value. For example, print("\101\102\103")
\xhh	Hex value. For example, print("\x41\x42\x43")

Remember: All string methods returns new values. They do not change the original string. Hence, string is immutable in python.

4. Boolean

Boolean variables represent one of two values: **True** or **False**. If we compare two values, the expression is evaluated and Python returns the Boolean answer. For example,

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

The ***bool()*** function allows you to evaluate any value, and give you True or False in return. For example,

```
print(bool("hello"))
print(bool(0))
```

Remember: Any string is True, except empty strings. Any number is True, except 0. Any list, tuple, set, and dictionary are True, except empty ones. The value None also evaluates to false.

5. Lists

Python lists are one of the most versatile data types that allow us to work with multiple elements at once. Lists are used to store multiple items in a single variable. Lists are created using square brackets. For example,

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

List items are *ordered*, *changeable*, and *allow duplicate values*. To determine how many items a list has, use the l**en()** function. List items can be of different data types. A single list can also contain different data types. For example,

```
list1 = ["apple", 24, "banana", 33, "cherry"]
print(len(list1))
```

The **type()** function is used to get the data type of the list. It is also possible to use the **list()** constructor when creating a new list. For example,

```
fruits = list(("apple", "banana", "cherry"))
```

- **Accessing list items:** We can use index number to access list items (first item has index 0, the second item has index 1, and so on; negative indexing means start from the end, -1 refers to the last item, -2 refers to the second last item and so on). For example,

  fruits = ["apple", "banana", "cherry"]
  fruits[0] = "grape"
  print(fruits[1])
  print(fruits[-2])

  You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items. For example,

  fruits = ["apple", "banana", "cherry"]
  print(fruits[1:3])

  By leaving out the start value, the range will start at the first item. By leaving out the end value, the range will go on to the end of the list. Specify negative indexes if you want to start the search from the end of the list. For example,

  thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
  print(thislist[:4])
  print(thislist[2:])
  print(thislist[-4:-1])

  We use **in** and **not in** operators to determine if a specified item is present in a list or not present in the list respectively.
- **Changing list items:** We use index number to change the value of a specific item and to change the value of items within a specific range, we use range of index numbers where we want to insert the new values. For example,

  fruits = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
  fruits[0] = "blueberry"
  fruits[1:3] = ["blackcurrant", "watermelon"]
  print(fruits)

  If we insert more items than you replace, the new items will be inserted where we specified, and the remaining items will move accordingly. For example,

  fruits = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
  fruits[1:2] = ["blackcurrant", "watermelon"]
  print(fruits)

  If we insert less items than we replace, the new items will be inserted where we specified, and the remaining items will move accordingly. For example,

  fruits = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
  fruits[1:4] = ["blackcurrant", "watermelon"]
  print(fruits)
- **Adding list items:** We use **insert()** method to insert a new list item, without replacing any of the existing values. This method inserts an item at the specified index. For example,

  fruits = ["apple", "banana", "cherry"]
  fruits.insert(1, "orange")

```
print(fruits)
```

we can append an item to the end of the list using **append()** method. We use **extend()** method to append elements from another list to the current list. For example,

```
fruits = ["apple", "banana", "cherry"]
drinks = ["coke", "pepsi"]
fruits.append("orange")
fruits.extend(drinks)
print(fruits)
```

- **Removing list items:** We use **remove()** method to remove specified item from the list. For example,

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)
```

To remove an item from specified index, we use **pop()** method. If we do not specify the index, the pop() method removes the last item. For example,

```
fruits.pop(1)
fruits.pop()
```

To delete specified index, we use **del** keyword. We can also use this keyword to delete the entire list. For example,

```
del fruits[1]
del fruits
```

The **clear()** method empties the list. The list still remains, but it has no content. For example,

```
fruits.clear()
```

- **Looping lists:** we can use **for** loop to loop through the list items. For example,

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

We can use **range()** and **len()** functions to loop through the list items by referring to their index number as given below.

```
fruits = ["apple", "banana", "cherry"]
for x in range(len(fruits)):
  print(fruits[x])
```

We can also use while loop to loop through the list items. For example,

```
fruits = ["apple", "banana", "cherry"]
x = 0
while x < len(fruits):
  print(fruits[x])
  x += 1
```

- **List comprehension:** List comprehension offers a shorter syntax (newlist = [*expression* **for** *item* **in** *iterable* **if** *condition*]) when you want to create a new list based on the values of an existing list. For example,

  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
  newlist = [x for x in fruits if "a" in x]
  print(newlist)

  In the above syntax, *condition* is like a filter that only accepts the items that evaluate to True. The condition is optional and can be omitted. The *iterable* can be any iterable object, like a list, tuple, set, range etc. The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list.
- **Sorting lists:** We can use **sort()** method of the list to sort the list in ascending order by default.

  fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
  fruits.sort()
  print(fruits)

  To sort the list in descending order, we use the keyword argument reverse = True. For example,

  fruits.sort(reverse = True)

  You can also customize your own function by using the keyword argument key = function. The function will return a number that will be used to sort the list (the lowest number first). For example,

  def myfunc(n):
  return abs(n - 50)
  thislist = [100, 50, 65, 82, 23]
  thislist.sort(key = myfunc)
  print(thislist)

  By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters. If we want a case-insensitive sort function, use str.lower as a key function. For example,

  thislist = ["banana", "Orange", "Kiwi", "cherry"]
  thislist.sort(key = str.lower)
  print(thislist)

  To reverse the order of list, we use **reverse()** method. For example,

  thislist.reverse()

- **Copying lists:** We can use **copy()** or **list()** method to copy a list into another list. For example,

  thislist = ["apple", "banana", "cherry"]
  mylist = thislist.copy() #mylist = list(thislist)
  print(mylist)

- **Joining lists:** We can join two lists using + operator or **append()** method or **extend()** method. For example,
  list1 = ["a", "b", "c"]
  list2 = [1, 2, 3]

list3 = list1 + list2
If you want to multiply the content of a liste a given number of times, you can use the * operator. For example,
list1 = ["a", "b", "c"]
*list2 = list1 * 2*
print(list2)

6. Tuples

Tuples are also used to store multiple items in a single variable. A tuple is a collection of items which is *ordered*, *unchangeable*, and *allow duplicates*. Tuples are written with round brackets. For example,

fruits = ("apple", "banana", "cherry", "banana")
print(fruits)

We use **len()** function to determine how many items a tuple has. For example, *len(fruits)*.
To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple. For example,

fruits = ("apple",)

Tuple items can be of different data types and a single tuple can also contain different data types. For example,

tuple1 = ("apple", 24, "banana", 33, "cherry")
print(tuple1)

The **type()** function is used to get the data type of the tuple. It is also possible to use the **tuple()** constructor when creating a new tuple. For example,

fruits = tuple(("apple", "banana", "cherry"))

- **Updating tuples:** Tuples are unchangeable or immutable, meaning that you cannot change, add, or remove items once the tuple is created. However, we can convert the tuple into a list, change the list, and convert the list back into a tuple. For example,

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

  You are allowed to add tuples to tuples, so if you want to add, create a new tuple with items, and add it to the existing tuple. For example,

  thistuple = ("apple", "banana", "cherry")
  y = ("orange","kiwi")
  thistuple += y
  print(thistuple)

  The **del** keyword can delete the tuple completely. For example, *del thistuple*.
- **Accessing tuple items:** We can use index number to access tuple items (first item has index 0, the second item has index 1, and so on; negative indexing

means start from the end, -1 refers to the last item, -2 refers to the second last item and so on). For example,

```
fruits = ("apple", "banana", "cherry")
print(fruits[1])
print(fruits[-2])
```

You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items. For example,

```
fruits = ("apple", "banana", "cherry")
print(fruits[1:3])
```

By leaving out the start value, the range will start at the first item. By leaving out the end value, the range will go on to the end of the list. Specify negative indexes if you want to start the search from the end of the tuple. For example,

```
thislist = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thislist[:4])
print(thislist[2:])
print(thislist[-4:-1])
```

We use **in** and **not in** operators to determine if a specified item is present in a list or not present in the list respectively.

- **Unpacking tuples:** Creating a tuple is called packing. Unpacking allows us to extract values back into variables. For example,

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)
```

The number of variables must match the number of values in the tuple, if not, you must use an asterisk (*) to collect the remaining values as a list. For example,

```
fruits = ("apple", "banana", "cherry")
(green, *yellow) = fruits
print(green)
print(yellow)
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left. For example,

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
(green, *tropic, red) = fruits
print(green)
print(tropic)
print(red)
```

- **Looping tuples:** we can use **for** loop to loop through the tuple items. For example,

```
fruits = ("apple", "banana", "cherry")
for x in fruits:
```

```
print(x)
```

We can use **range()** and **len()** functions to loop through the tuple items by referring to their index number as given below.

```
fruits = ("apple", "banana", "cherry")
for x in range(len(fruits)):
  print(fruits[x])
```

We can also use while loop to loop through the list items. For example,

```
fruits = ("apple", "banana", "cherry")
x = 0
while x < len(fruits):
  print(fruits[x])
  x += 1
```

- **Joining tuples:** We can join two tuples using **+** operator. For example,

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

If you want to multiply the content of a tuple a given number of times, you can use the * operator. For example,

```
tuple1 = ("a", "b", "c")
tuple2 = tuple1 * 2
print(tuple2)
```

8. Sets

Sets are also used to store multiple items in a single variable. A set is a collection of items which is *unordered, unchangeable* (but we and add and remove items), *unindexed*, and *do not allow duplicates*. Sets are written with curly brackets. For example,

```
fruits = {"apple", "banana", "cherry", "banana"}
print(fruits)
```

We use **len()** function to determine how many items a set has. For example, *len(fruits)*.
Set items can be of different data types and a single set can also contain different data types. For example,

```
set1 = {"apple", 24, "banana", 33, "cherry"}
print(set1)
```

The **type()** function is used to get the data type of the set. It is also possible to use the **set()** constructor when creating a new tuple. For example,

```
fruits = set(("apple", "banana", "cherry"))
```

- **Accessing sets:** We cannot access items in a set by referring to an index or a key. But we can loop through set items by using ***loops*** or the ***in*** keyword. For example,

```
fruits = {"apple", "banana", "cherry"}
```

```
for x in fruits:
  print(x)
print("banana" in fruits)
```

- **Adding items:** Once a set is created, we cannot change its items, but we can add new items using **add()** method. For example,

  ```
  fruits = {"apple", "banana", "cherry"}
  fruits.add("grapes")
  ```

  We can add items from another set into the current set by using **update()** method. For example,

  ```
  fruit1 = {"apple", "banana", "cherry"}
  fruit2 = {"pineapple", "mango", "papaya"}
  fruit1.update(fruit2)
  ```

  Remember: The object in the update() method does not have to be a set, it can be any iterable object such as tuples, lists, dictionaries etc.
- **Removing items:** We can remove items from a set using **remove()** or **discard()** method. If the item to remove does not exist, remove() will raise an error but discard() does not. For example,

  ```
  fruits = {"apple", "banana", "cherry"}
  fruits.remove("bananaa")
  print(fruits)
  ```

  We can also use **pop()** method to remove an item. Since sets are unordered, we will not know what item that gets removed. The return value of the pop() method is the removed item. For example,

  ```
  fruits.pop()
  ```

  We use clear() method to make set empty and del keyword to delete set completely. For example,

  ```
  fruits.clear()
  del fruits
  ```

- **Looping sets:** we can use **for** loop to loop through set items. For example,

  ```
  fruits = {"apple", "banana", "cherry"}
  for x in fruits:
    print(x)
  ```

- **Joining sets:** We can use **union()** or **update()** method to join two sets. The union() method returns new set containing all items from both sets and the update() method inserts all the items from one set into another. For example,

  ```
  set1 = {"a", "b", "c"}
  set2 = {1, 2, 3}
  set3 = set1.union(set2)
  #set1.update(set2)
  print(set3)
  ```

- **Keeping only duplicates:** We can use **intersection_update()** or **intersection()** method to keep only the items that are present in both sets. The *intersection()* method returns new set containing items that are present in both

sets and the *intersection_update()* method inserts common items from one set into another. For example,

```
set1 = {"a", "b" , "c", 3}
set2 = {1, "a", 3}
set3 = set1.intersection(set2)
#set1.intersection_update(set2)
print(set3)
```

Remember: The **symmetric_difference()** method will return a new set, that contains only the elements that are NOT present in both sets and the **symmetric_difference_update()** method will keep only the items in this set that are NOT present in both sets. The **difference()** method returns a set containtng the difference between two sets. And, the **difference_update()** method removes the items in this set that are also included in another set.

9. Frozenset

The **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable. Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation. For example,

```
fset = frozenset(['Ram', 'Shyam', 'Hari'])
print(fset)
print(type(fset))
```

Remember: Like normal sets, frozenset can also perform different set operations like copy, difference, intersection, symmetric_difference, union etc.

10. Range

The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops. The **range()** function allows the user to generate a series of numbers within a given range. Depending on how many arguments the user is passing to the function, the user can decide where that series of numbers will begin and end, as well as how big the difference will be between one number and the next. The syntax for range function is:

range(start, stop, step)

- *start: [optional] start value of the sequence; Default is 0*
- *stop: next value after the end value of the sequence*
- *step: [optional] integer value, denoting the difference between any two numbers in the sequence; Default is 1*

For example,

```
for x in range(2, 10, 2):
    print(x, end = " ")
```

11. Dictionary

Python dictionaries are used to store data in ***key:value*** pairs in a single variable. A dictionary is a collection of items which is *ordered* (In Python 3.6 and earlier, dictionaries are unordered), *changeable*, and *do not allow duplicates* (duplicate values

will overwrite existing values). Dictionaries are written with curly brackets. For example,

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

We use **len()** function to determine how many items a dictionary has. For example, *len(thisdict)*.
The values in dictionary items can be of any data type. The **type()** function is used to get the data type of the dictionary. For example, *type(thisdict)*. It is also possible to use the **dict()** constructor when creating a new dictionary. For example,

```
x = dict(name = "John", age = 36, country = "Norway")
```

- **Accessing items:** Dictionary items can be accessed to by using the key name. For example, *thisdict["brand"]*.
  We can also use **get()** method to access dictionary items. For example, *thisdict.get("brand")*.
  The **keys()** method will return a list of all the keys in the dictionary. For example, *thisdict.keys()*.
  The **values()** method will return a list of all the values in the dictionary. For example, *thisdict.values()*.
  The **items()** method will return each item in a dictionary, as tuples in a list. For example, *thisdict.items()*.
  We use **in** keyword to determine if a specified key is present in a dictionary. For example,

  ```
  thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
  }
  if "model" in thisdict:
     print("Key found")
  ```

  The **update()** method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs. For example,
  thisdict.update({"year": 2020})
- **Adding items:** Adding an item to the dictionary is done by using a new key and assigning a value to it. For example,

  ```
  thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
  }
  thisdict["color"] = "White"
  ```

The **update()** method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added. The argument must be a dictionary, or an iterable object with key:value pairs. For example,

thisdict.update({"color": "White"})

- **Removing items:** The **pop()** method removes the item with the specified key name. For example, *thisdict.pop("model")*.
  The **popitem()** method removes the last inserted item. For example, *thisdict.popitem()*.
  The ***del*** keyword removes the item with the specified key name. For example, *del thisdict["model"]*. The ***del*** keyword can also delete the dictionary completely. For example, *del thisdict*. The **clear()** method empties the dictionary. For example, *thisdict.clear()*.
- **Looping dictionary:** We can loop through a dictionary by using a for loop. When looping through a dictionary, the return value are the keys of the dictionary. For example,

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
   print(x)
```

We can also print all values in the dictionary, one by one as given below.

```
for x in thisdict:
   print(thisdict[x])
```

We can also use the **values()** method to return values of a dictionary. For example,

```
for x in thisdict.values():
   print(x)
```

We can use the **keys()** method to return the keys of a dictionary. For example,

```
for x in thisdict.keys():
   print(x)
```

To loop through both keys and values, by using the items() method. For example,

```
for x in thisdict.items():
   print(x)
```

- **Copying dictionary:** We can use copy() method to make a copy of a dictionary. For example,

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thatdict = thisdict.copy()
```

Another way to make a copy is to use the built-in function **dict()**. For example, *thatdict = dict(thisdict)*.

- **Nested dictionaries:** Like lists, we can create dictionaries that contain other directories as given below.

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
print(myfamily)
```