PYTHON

1. **Basics of Python**

1.1. Introduction

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

1.2. The print() Function

The print() function prints the specified message to the screen, or other standard output device.The message can be a string, or any other object, the object will be converted into a string before written to the screen.

> **NOTE :** The print() function displays the value inside its parentheses on the screen.

- **Example :**

```
print('Hello, world!')
```

1.3. The input() Function

The input() function waits for the user to type some text on the keyboard and press ENTER.

- **Example :**

```
name = input()
print('Hello! It is good to meet you, ' + myName)
```

1.4. Python Comments

Comments starts with a #, and Python will ignore them

- **Example :**

```
# This is a comment.
```

1.5. Python Indentation

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

- **Example :** Python uses indentation to indicate a block of code.

```
if 8 > 2:
  print("Eight is greater than two!")
```

2. **Python Variables**

2.1. Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

2.2. Rules for Python variables

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
    - **Legal variable names :**
        - myvar = "David"
        - my_var = "David"
        - _my_var = "David"
        - myVar = "David"
        - MYVAR = "David"
        - myvar2 = "David"
    - **Illegal variable names :**
        - 2myvar = "David"
        - my-var = "David"
        - my var = "David"

2.3. Python Scope

A variable is only available from inside the region it is created. This is called scope.

- **Local Scope :** A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.
- **Local Variable :** The local variable can be accessed from a function within the function :
    - **Example :**

```
def my_func():
  x = 50
  def my_innerfunc():
    print(x)
  my_innerfunc()
my_func()
```

- **Global Scope :** A variable created in the main body of the Python code is a global variable and belongs to the global scope.
- **Global Variable :** Global variables are available from within any scope, global and local.

- **Example :**

```
x = 50
def my_func():
  print(x)
my_func()
print(x)
```

3. **Python Keywords**

Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers.

3.1. Keyword Description

> - **and :** A logical operator
> - **as :** To create an alias
> - **assert :** For debugging
> - **break :** To break out of a loop
> - **class :** To define a class
> - **continue :** To continue to the next iteration of a loop
> - **def :** To define a function
> - **del :** To delete an object
> - **elif :** Used in conditional statements, same as else if
> - **else :** Used in conditional statements
> - **except :** Used with exceptions, what to do when an exception occurs
> - **False :** Boolean value, result of comparison operations
> - **finally :** Used with exceptions, a block of code that will be executed no matter if there is an exception or not
> - **for :** To create a for loop
> - **from :** To import specific parts of a module
> - **global :** To declare a global variable
> - **if :** To make a conditional statement
> - **import :** To import a module
> - **in :** To check if a value is present in a list, tuple, etc.
> - **is :** To test if two variables are equal
> - **lambda :** To create an anonymous function
> - **None :** Represents a null value
> - **nonlocal :** To declare a non-local variable
> - **not :** A logical operator
> - **or :** A logical operator
> - **pass :** A null statement, a statement that will do nothing
> - **raise :** To raise an exception
> - **return :** To exit a function and return a value
> - **True :** Boolean value, result of comparison operations
> - **try :** To make a try...except statement
> - **while :** To create a while loop
> - **with :** Used to simplify exception handling
> - **yield :** To end a function, returns a generator

4. **Python Data Types**

- Python has the following **`built-in data types`** by default, in these categories :
  - *`Text Type :`* str
  - *`Numeric Types :`* int, float, complex
  - *`Sequence Types :`* list, tuple, range
  - *`Set Types :`* set, frozenset
  - *`Mapping Type :`* dict
  - *`Boolean Type :`* bool
  - *`Binary Types :`* bytes, bytearray, memoryview

4.1. Getting the Datatype

Datatype of any object can be printed by using the type() function

- **Example :** Print the datatype of the variable x

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

5. **Python Strings**

[use single or double quotation] (for eg: my_name = "XYZ" or 'XYZ' both are same.)

5.1. Print string

```
print("Hello")
print('Hello')
```

5.2. Escape Characters

An escape character lets you use characters that are otherwise impossible to put into a string.

```
spam = 'Say hello to Riya\'s father.'
```

```
 (Escape character) |    (Print as)
 -------------------------------------------
     \'             |     Single quote
     \"             |     Double quote
     \t             |     Tab
     \n             |     Newline (line break)
     \\             |     Backslash
```

5.3. Raw Strings

Place an r before the beginning quotation mark of a string to make it a raw string. A raw string completely ignores all escape characters and prints any backslash that appears in the string.

```
print(r'That is David\'s bag.')
# output is : That is David\'s bag.
```

5.4. Putting Strings Inside Other Strings

```
name = 'David'
age = 35
print('Hello, my name is ' + name + '. I am ' + str(age) + ' years old.')
# output is :'Hello, my name is David. I am 35 years old.'
```

5.5. String interpolation

A simpler approach is to use string interpolation, in which the %s operator inside the string acts as a marker to be replaced by values following the string.

```
name = 'David'
age = 35
print('My name is %s. I am %s years old.' % (name, age))
# output is :'My name is David. I am 35 years old.'
```

5.6. f-strings

It is similar to string interpolation except that braces are used instead of %s, with the expressions placed directly inside the braces. Like raw strings, f-strings have an f prefix before the starting quotation mark.

```
name = 'David'
age = 35
print(f'My name is {name}. Next year I will be {age + 1}.')
# output is : 'My name is David. Next year I will be 36.'
```

5.7. Assign string to variable

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

5.8. Multiline string

```
a = """It is the simplest method
      to let a long string split into different lines.
      You will need to enclose it
      with a pair of Triple quotes"""
print(a)
```

5.9. Slicing the string

```
b = "Hello, World!"
print(b[2:5])
```

5.10. Negative indexing

```
b = "Hello, World!"
print(b[-4:-2])
```

5.11. String length

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

5.12. Check string

```
txt = "This is a method to check the presence of some string."
x = "ence" in txt
print(x)
txt = "This is a method to check the presence of some string."
x = "ence" not in txt
print(x)
```

5.13. String concatination

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

5.14. String format()

The format() method allows you to format selected parts of a string.

- **Use the format() method to insert numbers into string :**
  The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are

  ```
  age = 34
  txt = "My name is Nik, and I am {}"
  print(txt.format(age))
  ```

- **Format the price to be displayed as a number with two decimals**

  ```
  price = 50
  txt = "The price is {:.2f} dollars"
  print(txt.format(price))
  ```

- **Multiple Values :**
  If you want to use more values, just add more values to the format() method

  ```
  quantity = 5
  itemno = 127
  price = 50
  ```

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

- **Index Numbers :**
  You can use index numbers to be sure the values are placed in the correct placeholders

```
age = 40
name = "David"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name))
```

- *You can also use named indexes by entering a name inside the curly brackets {carname}, but then you must use names when you pass the parameter values txt.format(carname = "Ford")*

```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname = "Ford", model = "Mustang"))
```

5.15. **Useful String Methods**

```
# strip()
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!". It removes spaces outside string.

# lower()
a = "Hello, World!"
print(a.lower())

# upper()
a = "Hello, World!"
print(a.upper())

# replace()
a = "Hello, World!"
print(a.replace("W", "J"))

# split()
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']

# join() : The join() method is useful when you have a list of strings that need to be
joined together into a single string value.

print(', '.join(['cats', 'rats', 'bats']))
  # output is : 'cats, rats, bats'

print(' '.join(['My', 'name', 'is', 'Kevy']))
  # output is : 'My name is Kevy'
```

```
print('ABC'.join(['My', 'name', 'is', 'Kevy']))
  # output is : 'MyABCnameABCisABCKevy'
```

6. **Python Numbers**

There are three numeric types in Python :

- **int**
- **float**
- **complex**
    - Variables of numeric types are created when you assign a value to them :
        - **Example :**

```
x = 1    # int
y = 2.8  # float
z = 1+1j   # complex
```

7. **Python Lists**

[A list is a collection which is ordered and changeable. In Python lists are written with square brackets.]

7.1. Create list

```
list = ["apple", "orange", "cherry"]
print(list)
```

7.2. Access items

```
list = ["apple", "orange", "cherry"]
print(list[1])
```

7.3. Negative indexing

```
list = ["apple", "orange", "cherry"]
print(list[-1])
```

7.4. Range

```
list = ["apple", "orange", "cherry", "banana", "kiwi", "melon", "mango"]
print(list[2:5])
```

> **NOTE :** The search will start at index 2 (included) and end at index 5 (not included).

- By leaving out the start value, the range will start at the first item:

  ```
  list = ["apple", "orange", "cherry", "banana", "kiwi", "melon", "mango"]
  print(list[:4])
  ```

- By leaving out the end value, the range will go on to the end of the list :

  ```
  list = ["apple", "orange", "cherry", "banana", "kiwi", "melon", "mango"]
  print(list[2:])
  ```

7.5. Change Item Value

```
list = ["apple", "orange", "cherry"]
list[1] = "blackcurrant"
```

```
print(list)
```

7.6. Loop Through a List

You can loop through the list items by using a for loop:

```
list = ["apple", "orange", "cherry"]
for x in list:
    print(x)
```

7.7. Check if Item Exists

To determine if a specified item is present in a list use the in keyword

```
list = ["apple", "orange", "cherry"]
if "apple" in list:
    print("Yes, 'apple' is in the fruits list")
```

7.8. Check Length

```
list = ["apple", "orange", "cherry"]
print(len(list))
```

7.9. List Methods with examples

```
# Using the append() method to append an item :
list = ["apple", "orange", "cherry"]
list.append("banana")
print(list)

# To add an item at the specified index, use the insert() method :
list = ["apple", "orange", "cherry"]
list.insert(1, "banana")
print(list)

# The remove() method removes the specified item :
list = ["apple", "orange", "cherry"]
list.remove("orange")
print(list)

# The pop() method removes the specified index, (or the last item if index is not
specified):
list = ["apple", "orange", "cherry"]
list.pop()
print(list)
```

```
# The del keyword removes the specified index:
list = ["apple", "orange", "cherry"]
del list[0]
print(list)

# The clear() method empties the list:
list = ["apple", "orange", "cherry"]
list.clear()
print(list)

# copy the list
list = ["apple", "orange", "cherry"]
mylist = list.copy()
print(mylist)

# Make a copy of a list with the list() method:
list = ["apple", "orange", "cherry"]
mylist = list(list)
print(mylist)

# Join two list
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)

# Append list2 into list1:
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
for x in list2:
  list1.append(x)
print(list1)

# Use the extend() method to add list2 at the end of list1:**
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
```

7.10. **Useful Lists methods**

- **append()** Adds an element at the end of the list
- **clear()** Removes all the elements from the list
- **copy()** Returns a copy of the list
- **count()** Returns the number of elements with the specified value
- **extend()** Add the elements of a list (or any iterable), to the end of the current list
- **index()** Returns the index of the first element with the specified value
- **insert()** Adds an element at the specified position
- **pop()** Removes the element at the specified position
- **remove()** Removes the item with the specified value
- **reverse()** Reverses the order of the list

- **sort()** Sorts the list

8. **Python Tuples**

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

- **Example :**

```
tuple = (15,"xyz", 1.5)
print(tuple[0])
# returns 15
```

8.1. Create a Tuple

```
tuple = ("apple", "orange", "cherry")
print(tuple)
```

> **NOTE :** Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

8.2. Convert the tuple into a list to be able to change it

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

8.3. Add Items

Once a tuple is created, you cannot add items to it. Tuples are unchangeable.You cannot add items to a tuple

```
tuple = ("apple", "orange", "cherry")
tuple[3] = "banana" # This will raise an error
print(tuple)
```

8.4. Create Tuple With One Item

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.One item tuple, remember the commma

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

8.5. NOT a tuple

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

8.6. Remove Items

> **NOTE :** You cannot remove items in a tuple.Tuples are unchangeable, so you cannot remove items from it, but you can delete the tuple completely.

8.7. The del keyword can delete the tuple completely

```
tuple = ("apple", "orange", "cherry")
del tuple
print(tuple) #this will raise an error because the tuple no longer exists
```

8.8. Count the number of items

Return the number of times the value 5 appears in the tuple:

```
tuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = tuple.count(5)
print(x)
```

9. **Python Sets**

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

9.1. Create a Set

```
set = {"apple", "orange", "cherry"}
print(set)
```

> **NOTE :** Sets are unordered, so you cannot be sure in which order the items will appear. Change Items-Once a set is created, you cannot change its items, but you can add new items. If the item to remove does not exist, remove() will raise an error.If the item to remove does not exist, discard() will NOT raise an error.Sets are unordered, so when using the pop() method, you will not know which item that gets removed.The union() method returns a new set with all items from both sets.The update() method inserts the items in set2 into set1:Both union() and update() will exclude any duplicate items.

9.2. **Useful Set Methods**

- **add()** Adds an element to the set
- **clear()** Removes all the elements from the set
- **copy()** Returns a copy of the set
- **difference()** Returns a set containing the difference between two or more sets
- **difference_update()** Removes the items in this set that are also included in another, specified set
- **discard()** Remove the specified item
- **intersection()** Returns a set, that is the intersection of two other sets
- **intersection_update()** Removes the items in this set that are not present in other, specified set(s)
- **isdisjoint()** Returns whether two sets have a intersection or not
- **issubset()** Returns whether another set contains this set or not
- **issuperset()** Returns whether this set contains another set or not
- **pop()** Removes an element from the set
- **remove()** Removes the specified element
- **symmetric_difference()** Returns a set with the symmetric differences of two sets
- **symmetric_difference_update()** inserts the symmetric differences from this set and another
- **union()** Return a set containing the union of sets
- **update**() Update the set with the union of this set and others

10. **Python Dictionaries**

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

10.1. Create and print a dictionary

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

10.2. Some dictionaries methods with examples

```
# Get the value of the "model" key:
    x = dict["model"]

# Change the "year" to 2018:
    dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
    }
    dict["year"] = 2018

# Print all key names in the dictionary, one by one:
    for x in dict:
        print(x)

# Print all key names in the dictionary, one by one:
    for x in dict:
        print(x)

# You can also use the values() function to return values of a dictionary:
    for x in dict.values():
        print(x)

# Loop through both keys and values, by using the items() function:
    for x, y in dict.items():
        print(x, y)

# Adding Items : Adding an item to the dictionary is done by using a new index key and
assigning a value to it:
    dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
    }
    dict["color"] = "red"
    print(dict)

# The pop() method removes the item with the specified key name:
    dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
    }
    dict.pop("model")
    print(dict)

# The del keyword removes the item with the specified key name:
    dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
    }
    del dict["model"]
    print(dict)

# The clear() method empties the dictionary:
    dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
    }
    dict.clear()
    print(dict)
```

10.3. **Useful Dictionaries Methods**

- **clear()** Removes all the elements from the dictionary
- **copy()** Returns a copy of the dictionary
- **fromkeys()** Returns a dictionary with the specified keys and value
- **get()** Returns the value of the specified key
- **items()** Returns a list containing a tuple for each key value pair
- **keys()** Returns a list containing the dictionary keys
- **pop()** Removes the element with the specified key
- **popitem()** Removes the last inserted key-value pair
- **setdefault()** Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
- **update()** Updates the dictionary with the specified key-value pairs
- **values()** Returns a list of all the values in the dictionary

11. **Python Booleans**

Booleans represent one of two values : True or False. In programming you often need to know if an expression is True or False. You can evaluate any expression in Python, and get one of two answers, True or False. When you compare two values, the expression is evaluated and Python returns the Boolean answer

```
print(10 > 9) # output is True
print(10 == 9) # output is False
print(10 < 9) # output is False
```

12. **Python Conditions and If statements**

12.1. If statement

~~~
```
a = 51
b = 210
if b > a:
  print("b is greater than a")
```
~~~

12.2. Elif

The elif keyword is pythons way of saying 'if the previous conditions were not true, then try this condition'

```
a = 15
b = 15
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

12.3. Else

The else keyword catches anything which isn't caught by the preceding conditions.

```
a = 210
b = 51
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

12.4. Nested If

You can have if statements inside if statements, this is called nested if statements.

```
x = 62
if x > 10:
  print("Above ten,")
if x > 20:
  print("and also above 20!")
```

```
else:
  print("but not above 20.")
```

12.5. The pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 51
b = 210
if b > a:
  pass
```

13. **Python Loops**

Python has two primitive loop commands

- ***while loops***
- ***for loops***

13.1. **The while Loop**

With the while loop we can execute a set of statements as long as a condition is true.

- **Example :**
  Print i as long as i is less than 6

```
i = 1
while i < 9:
  print(i)
  i += 1
```

> **NOTE :** remember to increment i, or else the loop will continue forever.

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

13.1.1. The break Statement

With the break statement we can stop the loop even if the while condition is true

- **Example :** Exit the loop when i is 3

```
i = 1
while i < 9:
  print(i)
  if i == 4:
    break
  i += 1
```

13.1.2. The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

- **Example :**
  Continue to the next iteration if i is 3

```
i = 0
while i < 9:
  i += 1
```

```
    if i == 4:
      continue
    print(i)
```

13.1.3. The else Statement

With the else statement we can run a block of code once when the condition no longer is true

- **Example :**
  Print a message once the condition is false:

```
i = 1
while i < 9:
  print(i)
  i += 1
else:
  print("i is no longer less than 9")
```

13.2. **The For Loops**

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

- **Example :**
  Print each fruit in a fruit list:

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

> NOTE : The for loop does not require an indexing variable to set beforehand.

13.2.1. Looping Through a String

Even strings are iterable objects, they contain a sequence of characters

- **Example :**
  Loop through the letters in the word "orange":

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

13.2.2. The break Statement

With the break statement we can stop the loop before it has looped through all the items

- **Example :**
  Exit the loop when x is "orange":

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

- **Example :**
  Exit the loop when x is "orange", but this time the break comes before the print:

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

13.2.3. The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next

- **Example :**
  Do not print orange :

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

13.2.4. The range() Function

To loop through a set of code a specified number of times, we can use the range() function,
The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

- **Example :**
  Using the range() function

```
for x in range(7):
  print(x)
```

> **NOTE :** range(7) is not the values of 0 to 7, but the values 0 to 6.

- The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 7), which means values from 2 to 7 (but not including 7)
- **Example :** Using the start parameter

```
for x in range(2, 6):
  print(x)
```

- The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 40, 2)
- **Example :** Increment the sequence with 2 (default is 1)

```
for x in range(2, 40, 2):
  print(x)
```

13.2.5. Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished

- **Example :** Print all numbers from 0 to 6, and print a message when the loop has ended:

```
for x in range(7):
  print(x)
else:
  print("Its finished!")
```

13.2.6. Nested Loops

A nested loop is a loop inside a loop.The "inner loop" will be executed one time for each iteration of the "outer loop"

- **Example :** Print each adjective for every fruit:

```
adj = ["green", "big", "tasty"]
fruits = ["apple", "orange", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```

13.2.7. The pass Statement

for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

- **Example :**

```
for x in [0, 1, 2]:
  pass
```

14. **Python Functions**

A function is a block of code which only runs when it is called.

> **NOTE :** The terms parameter and argument can be used for the same thing: information that are passed into a function.
> From a function's perspective:
>
> - A parameter is the variable listed inside the parentheses in the function definition.
> - An argument is the value that is sent to the function when it is called.

14.1. Creating a Function

In Python a function is defined using the def keyword

- **Example** :

```
def my_func():
  print("Hello! This is a function")
```

14.2. Calling a Function

To call a function, use the function name followed by parenthesis

- **Example :**

```
def my_func():
  print("Hello! This is a function")
my_func()
```

14.3. Arguments

Information can be passed into functions as arguments.

- **Example :**

```
def my_func(name):
  print("Hello" + name)
my_func("David")
my_func("Savi")
my_func("Navi")
```

- **Number of Arguments :** By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.
  - Example : This function expects 2 arguments, and gets 2 arguments:

```
def my_func(name, country):
  print(name + " " + country)
my_func("David", "America")
```

- **Arbitrary Arguments,** *args : If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
    - Arbitrary Arguments are often shortened to *args in Python documentations.

```
def my_func(*kids):
  print("The youngest child is " + kids[2])
my_func("David", "Nik", "Linus")
```

- **Keyword Arguments :** You can also send arguments with the key = value syntax.This way the order of the arguments does not matter.
    - The phrase Keyword Arguments are often shortened to kwargs in Python documentations.

```
def my_func(child3, child2, child1):
  print("The youngest child is " + child3)
my_func(child1 = "David", child2 = "Nik", child3 = "Linus")
```

- **Arbitrary Keyword Arguments**, **kwargs : If the number of keyword arguments is unknown, add a double ** before the parameter name:
    - Arbitrary Kword Arguments are often shortened to **kwargs in Python documentations.

```
def my_func(**kid):
  print("His last name is " + kid["lname"])
my_func(fname = "Nik", lname = "jons")
```

- **Return Values :** To let a function return a value, use the return statement:
    - Example :

```
def my_func(x):
  return 5 * x
print(my_func(3))
print(my_func(5))
print(my_func(9))
```

15. **Python Lambda**

A lambda function is a small anonymous function.
A lambda function can take any number of arguments, but can only have one expression.

- **Syntax -**
  lambda arguments : expression
- **The expression is executed and the result is returned:**
  - **Example :**
    A lambda function that adds 10 to the number passed in as an argument, and print the result

```
x = lambda a : a + 20
print(x(5))
```

15.1. Lambda functions can take any number of arguments

- **Example :**
  A lambda function that multiplies argument a with argument b and print the result ```
  x = lambda a, b : a * b
  print(x(7, 3)

```
```

- **Example :** A lambda function that sums argument a, b, and c and print the result:

```
x = lambda a, b, c : a + b + c
print(x(7, 4, 2))
```

16. **Python Map Function**

The map() function is a built-in function.

- **Syntax :**
  map(function, iterable [, iterable2, iterable3,...iterableN]) --> map object

The map() function calls the specified function for each item of an iterable (such as string, list, tuple or dictionary) and returns a list of results.

- **Consider the following simple square function :**

```
def square(x):
  return x*x
```

16.1. map function with the list of numbers to get the list of results

```
def square(x):
  return x*x

numbers = [1, 2, 3, 4, 5]

sqList = map(square, numbers)

print(next(sqrList))

print(next(sqrList))

print(next(sqrList))

print(next(sqrList))

print(next(sqrList))
```

In the above example, the map() function applies to each element in the numbers[] list. This will return a map object which is iterable and so, we can use the next() function to traverse the list.

16.2. Map with Lambda Expression

The map() function passes each element in the list to the built-in function, a lambda function or a user-defined function, and returns the mapped object. The following map() is used with the lambda function.

```
sqrList = map(lambda x: x*x, [1, 2, 3, 4])

print(next(sqrList))

print(next(sqrList))
print(next(sqrList))

print(next(sqrList))
```

17. **Python classes and objects**

Python is an object oriented programming language which stress on objects.

Object *is simply a collection of data (variables) and methods (functions) that act on those data and* **Class** *is a blueprint for the object.*

17.1. Create a Class

A class is defined by using the keyword class.

- **Example :**
  Create a class named MyClass

```
class MyClass:
  x = 7
```

17.2. Create Object

Now we can use the class named MyClass to create objects.

- **Example :**
  Create an object named p1, and print the value of x:

```
ob = MyClass()
print(ob.x)
```

17.3. The `__init__()` Function

All classes have a function called `__init__()`, which is always executed when the class is being initiated.This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

- **Example :**
  Create a class named student, use the `__init__()` function to assign values for name and id.

```
class student:
  def __init__(self, name, id):
    self.name = name
    self.id = id

ob = student("Aman", 326)

print(ob.name)
print(ob.id)
```

> **NOTE :** The `__init__()` function is called automatically every time the class is being used to create a new object.

17.4. Object Methods

Methods in objects are functions that belong to the object.

- **Example :**
  Insert a function that prints a greeting, and execute it on the ob object.

```
class student:
  def __init__(self, name, id):
    self.name = name
    self.id = id

  def myfunc(self):
    print("Hello my name is " + self.name)

ob = student("Ammy", 10766)
ob.myfunc()
```

> **NOTE :** The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

17.5. self Parameter

Whenever an object calls its method, the object itself is passed as the first argument. So, ob.func() translates into MyClass.func(ob). The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class. It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.

- **Example :**
  Use the words xyz and abc instead of self:

```
class student:
  def __init__(xyz, name, id):
    xyz.name = name
    xyz.id = id

  def myfunc(abc):
    print("Hello my name is " + abc.name)

ob = student("Ammy", 10706)
ob.myfunc()
```

17.6. **Python Inheritance**

Inheritance allows us to define a class that inherits all the methods and properties from another class.

- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.

17.6.1. Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class.

- **Example :**
  Create a class named student, with firstname and lastname properties, and a printname method:

```
class student:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

ob = student("Jim", "Shang")
ob.printname()
```

17.6.2. Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class.

- **Example :**
  Create a class named children, which will inherit the properties and methods from the student class:

```
class children(student):
  pass
```

```
# Now the Student class has the same properties and methods as the Person class.
```

- **Example :**
  Use the Student class to create an object, and then execute the printname method:

```
class student:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class children(student):
  pass
```

```
x = children("Jim", "Shang")
x.printname()
```

17.6.3. Add the `__init__()` Function

Add the `__init__()` function to the child class (instead of the pass keyword).

> **NOTE :** The `__init__()` function is called automatically every time the class is being used to create a new object.

- **Example :**
  Add the `__init__()` function to the children class

```
class children(student):
  def __init__(self, fname, lname):
    #add properties etc.
```

> **NOTE :** When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function. The child's `__init__()` function overrides the inheritance of the parent's `__init__()` function. To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function.

- **Example :**

```
class student:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class children(student):
  def __init__(self, fname, lname):
    student.__init__(self, fname, lname)

x = children("Jim", "Shang")
x.printname()
```

17.6.4. super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent.

- **Example :**

```
class student:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class children(student):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)

x = children("Jim", "Shang")
x.printname()
```

17.6.5. Add Properties

- **Example :**
  Add a property called graduationyear to the Student class:

```
class student:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class children(student):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.identity = id

x = children("Jim", "Shang",10706)
x.identity()
```

18. **PIP**

PIP is a package manager for Python packages, or modules if you like.

> **NOTE :** If you have Python version 3.4 or later, PIP is included by default.

18.1. Check if PIP is Installed

Navigate your command line to the location of Python's script directory, and type the following

```
pip --version
```

18.2. Install PIP

If you do not have PIP installed, you can download and install it from this page: https://pypi.org/project/pip/

18.3. Download a Package

Navigate your command line to the location of Python's script directory, and type the following

```
pip install <package name>
```

Now you have downloaded and installed your first package!

18.4. Find Packages

Find more packages at https://pypi.org/.

18.5. Remove a Package

Use the uninstall command to remove a package.

```
pip uninstall <package name>
```

The PIP Package Manager will ask you to confirm that you want to remove the package:
Proceed (y/n)?
Press y and the package will be removed.

18.6. List Packages

Use the list command to list all the packages installed on your system.

```
pip list
```

19. **Python Modules**

A module is a file containing a set of functions you want to include in your application.

19.1. Create a Module

To create a module just save the code you want in a file with the file extension .py

- **Example :**
  Save this code in a file named myModule.py

```
def greeting(name):
  print("Hello, " + name)
```

19.2. Use a Module

Now we can use the module we just created, by using the import statement

- **Example :**
  Import the module named myModule, and call the greeting function:

```
import myModule
myModule.greeting("David")
```

> **NOTE :** When using a function from a module, use the syntax: module_name.function_name.

19.3. Variables in Module

The module can contain functions and also variables of all types (arrays, dictionaries, objects etc)

- **Example :**
  Save this code in the file myModule.py

```
employee1 = {
  "name": "David",
  "age": 40,
  "country": "India"
}
```

- **Example :**
  Import the module named myModule, and access the employee1 dictionary :

```
import myModule
```

```
a = myModule.employee1["age"]
print(a)
```

19.4. Naming a Module

You can name the module file whatever you like, but it must have the file extension .py

19.5. Re-naming a Module

You can create an alias when you import a module, by using the 'as' keyword:

- **Example :**
  Create an alias for myModule called mx:

  ```
  import myModule as md

  a = md.employee1["age"]
  print(a)
  ```

19.6. **Some built-in Modules in Python**

19.6.1. **Python - os Module**

It is possible to automatically perform many operating system tasks. The OS module in Python provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory, etc.

- **Creating Directory :**
  We can create a new directory using the mkdir() function from the OS module.

  ```
  import os
  os.mkdir("d:\\tempdir")
  ```

- **Changing the Current Working Directory :**
  We must first change the current working directory to a newly created one before doing any operations in it. This is done using the chdir() function.

  ```
  import os
  os.chdir("d:\\tempdir")
  ```

- **Removing a Directory :**
  The rmdir() function in the OS module removes the specified directory either with an absolute or relative path.

  ```
  import os
  os.rmdir("tempdir")
  ```

19.6.2. **Python - sys Module**

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment.

- **sys.argv :**
  sys.argv returns a list of command line arguments passed to a Python script. The item at index 0 in this list is always the name of the script. The rest of the arguments are stored at the subsequent indices.

```
import sys
print(sys.argv)
```

- **sys.exit :**
  This causes the script to exit back to either the Python console or the command prompt. This is generally used to safely exit from the program in case of generation of an exception.

```
import sys
sys.exit
```

- **sys.maxsize :**
  Returns the largest integer a variable can take.

```
import sys
print(sys.maxsize)
```

- **sys.version :**
  This attribute displays a string containing the version number of the current Python interpreter.

```
import sys
print(sys.version)
```

19.6.3. **Python - datetime Module**

import a module named datetime to work with dates as date objects.

- **Import the datetime module and display the current date :**

```
import datetime

x = datetime.datetime.now()
print(x)
```

- **Creating Date Objects :**
  To create a date, we can use the datetime() class (constructor) of the datetime module. The datetime() class requires three parameters to create a date : year, month, day.

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

- **The strftime() Method :**
  The datetime object has a method for formatting date objects into readable strings.
  The method is called strftime(), and takes one parameter, format, to specify the format of the returned string:
  - **Example :**
    Display the name of the month:

    ```
    import datetime

    x = datetime.datetime(2018, 6, 1)

    print(x.strftime("%B"))
    ```

20. **Python Regular Expression**

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

20.1. Import the re module

Python has a built-in package called re, which can be used to work with Regular Expressions.

```
import re
```

20.2. RegEx in Python

When you have imported the re module, you can start using regular expressions.

- **Example :**
  Search the string to see if it starts with "The" and ends with "challenge". In Python a regular expression search is typically written as `match = re.search(pat, str)`

  ```
  import re

  txt = "The grat Dance challenge"
  x = re.search("^The.*challenge$", txt)
  ```

RegEx Functions

> - **re.search :** Check if given pattern is present anywhere in input string
> - **re.compile :** Compile a pattern for reuse, outputs re.Pattern object
> - **re.sub :** search and replace
> - **re.sub**(r'pat', f, s) : function f with re.Match object as argument
> - **re.escape :** automatically escape all metacharacters
> - **re.split :** split a string based on RE
> - **re.findall :** returns all the matches as a list
> - **re.finditer :** iterator with re.Match object for each match
> - **re.subn :** gives tuple of modified string and number of substitutions

21. **Python Random Module**

21.1. random.random()

Generates a random float number between 0.0 to 1.0. The function doesn't need any arguments.

```
import random
print(random.random())
```

21.2. random.randint()

Returns a random integer between the specified integers.

```
import random
print(random.randint(1,150))
```

21.3. random.randrange()

Returns a randomly selected element from the range created by the start, stop and step arguments. The value of start is 0 by default. Similarly, the value of step is 1 by default.

```
import random
print(random.randrange(1,10))
print(random.randrange(1,20,2))
print(random.randrange(0,200,10))
```

21.4. random.choice()

Returns a randomly selected element from a non-empty sequence. An empty sequence as argument raises an IndexError.

```
import random
print(random.choice('computer'))
print(random.choice([22,15,24,45,56,57]))
print(random.choice((22,15,24,45,56,57)))
```

21.5. random.shuffle()

This functions randomly reorders the elements in a list.

```
numbers=[22,15,24,45,56,57]
random.shuffle(numbers)
print(numbers)
```

22. **Python Requests Module**

The requests module allows you to send HTTP requests using Python.
The HTTP request returns a Response Object with all the response data (content, encoding, status, etc).

- **Example :**
  Make a request to a web page, and print the response text:

  ```
  import requests

  x = requests.get('https://example.com')

  print(x.text)
  ```

22.1. Download and Install the Requests Module

Navigate your command line to the location of PIP, and type the following:

```
pip install requests
```

- **Syntax :** requests.methodname(params)

22.2. **Method Description**

> - **delete(url, args) :** Sends a DELETE request to the specified url
> - **get(url, params, args) :** Sends a GET request to the specified url
> - **head(url, args) :** Sends a HEAD request to the specified url
> - **patch(url, data, args) :** Sends a PATCH request to the specified url
> - **post(url, data, json, args) :** Sends a POST request to the specified url
> - **put(url, data, args) :** Sends a PUT request to the specified url
> - **request(method, url, args) :** Sends a request of the specified method to the specified url

23. **Python Package**

A module can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules. As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

> **NOTE :** A directory must contain a file named **init**.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

- **Let's create a package named mypackage, using the following steps :**
  1. Create a new folder named D:\MyApp.
  2. Inside MyApp, create a subfolder with the name 'mypackage'.
  3. Create an empty **init**.py file in the mypackage folder.
  4. Using a Python-aware editor like IDLE, create modules greet.py and functions.py with following code:
- greet.py

```
def SayHello(name):
  print("Hello " + name)
return
```

- functions.py

```
def sum(x,y):
  return x+y
def average(x,y):
  return (x+y)/2
def power(x,y):
  return x**y
```

23.1. Importing a Module from a Package

Now, to test our package, invoke the Python prompt from the MyApp folder.
> D:\MyApp>python

- **Example :** Import the functions module from the mypackage package and call its power() function.

```
from mypackage import functions
functions.power(3,2)
OUTPUT : 9
```

- It is also possible to import specific functions from a module in the package

```
from mypackage.functions import sum
sum(10,20)
```

```
OUTPUT : 30
```

24. **Python Iterator**

Iterators are implicitly used whenever we deal with collections of data types such as list, tuple or string (they are quite fittingly called iterables). The usual method to traverse a collection is using the for loop, as shown below.

- **Example :**

```
List = [1, 2, 3, 4]
for item in List:
  print(item)
```

24.1. iter()

Instead of using the for loop as shown above, we can use the iterator function iter(). An iterator is an object which represents a data stream. It returns one element at a time. Python's built-in method iter() receives an iterable and returns an iterator object. The iterator object uses the `__next__()` method. Every time it is called, the next element in the iterator stream is returned. When there are no more elements available, StopIteration error is encountered.

```
list = [1, 2, 3]
it = iter(list)
print(it.__next__())
print(it.__next__())
print(it.__next__())
StopIteration
```

24.2. next()

The built-in function next() accepts an iterator object as a parameter and calls the `__next__()` method internally. Hence, `it.__next__()` is the same as next(it). The following example uses the next() method to iterate the list.

```
list = [1, 2, 3]
it = iter(list)
print(next(it))
print(next(it))
print(next(it))
StopIteration
```

25. **Python Generator**

Python provides a generator to create your own iterator function. A generator is a special type of function which does not return a single value, instead it returns an iterator object with a sequence of values. In a generator function, a yield statement is used rather than a return statement. The following is a simple generator function.

25.1. Generator Function

```
def myGen():
  print('First number')
  yield 10

  print('Second number')
  yield 20

  print('Last number')
  yield 30
```

NOTE : The difference between yield and return is that yield returns a value and pauses the execution while maintaining the internal states, whereas the return statement returns a value and terminates the execution of the function.

26. **Python Decorators**

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

26.1. A simple decorator

decorator is a callable that takes a callable as input and returns another callable.

- **Example :** The simplest decorator in which null_decorator is a callable (it's a function), it takes another callable as its input, and it returns the same input callable without modifying it.

```
def null_decorator(func):
  return func
```

26.2. Decorate (or wrap) another function

I've defined a greet function and then immediately decorated it by running it through the null_decorator function.

```
def null_decorator(func):
  return func
def greet():
  return 'Hello!'

greet = null_decorator(greet)

x = greet()
print (x)
```

26.3. Another syntax for decorator

Instead of explicitly calling null_decorator on greet and then reassigning the greet variable, you can use Python's @ syntax for decorating a function in one step:

```
def null_decorator(func):
  return func

@null_decorator

def greet():
  return 'Hello!'
greet()
```

26.4. Decorators Can Modify Behavior

here is decorator which converts the result of the decorated function to uppercase letters:

```
def uppercase(func):
  def wrapper():
    original_result = func()
    modified_result = original_result.upper()
    return modified_result
  return wrapper

@uppercase

def greet():
  return'Hello!'
x = greet()
print(x)
```

26.5. Applying Multiple Decorators to a Single Function

```
def strong(func):
  def wrapper():
    return '<strong>' + func() + '</strong>'
  return wrapper

def emphasis(func):
  def wrapper():
    return '<em>' + func() + '</em>'
  return wrapper

@strong
@emphasis

def greet():
  return 'Hello!'
x = greet()
print(x)
```