👥

HARSHITH’S DATA SCIENCE BOOK

Python:

Python is a high-level, object-oriented, interpreted language that can be applied to various fields such as web development, machine learning, data analysis, etc.

Why Python:

- Python is known for its straightforward syntax and ease of use, making it a popular choice for beginners.
- With over 127,000 libraries available, it offers a vast range of possibilities for developers to work with.
- Python is widely used across various domains such as web development, machine learning, artificial intelligence, and more.
- Python is a Dynamic Language.

Variables:

Variables are named places in memory where values can be stored for later use.

```
a=10
b="Harshith"
```

In the above statement "a = 10 and b = "Harshith", "a" and "b" are examples of variables that are assigned the values 10 and "Harshith", respectively.

Rules of Declaring a Variable:

There are certain rules while declaring variables such as:

- Variable names shouldn’t start with a Number.
- There shouldn’t be gaps while declaring a variable.
- Python, variables are case-sensitive, meaning that "Name" and "name" are considered two different variables.

```
3name=10   #This is not a correct way
Id no=1739 #This is not a correct way

#Instead we can use the below methods

name3=10   #This is a correct way
Id_no=1739 #This is a correct way
```

Data Types:

Data types are the classification of data based on the type of value they hold. Some common data types in programming include:

1. Integer: a whole number
2. Float: a number with a decimal point
3. String: a sequence of characters (e.g. "Hello World")
4. Boolean: a data type that can have only two values, either True or False

5. Complex: which are in the format of a+bj
6. List: an ordered collection of values
7. Tuple: an ordered, immutable collection of values
8. Dictionary: an unordered collection of key-value pairs
9. set: an unordered collection of values

```
#Examples of declaring various basic Data types

#Integers
a=10
b=11339

#Float
a=1.22
nass=0.40
g=1.00

#Strings
a="ahhssh"
b="asdfghjkl"

#Boolean
a=True
b=False

#complex
a=2+3j
b=44-9j
```

Data Type Check:

In python, we can check the Data Type of the data using the inbuilt function type()

```
ip: type(123)
op: int

ip: type("ASDFFF")
op: str

ip: type(1.00)
op: float

ip: type(True)
op: boolean

ip: type(1+2j)
op: complex
```

Type Casting:

Type casting is the technique by which we can change the data type of the data from one data type to another.

For eg: if a=123 and it is an integer, we can easily change its Data Type by using float(a) or str(a). Here float value will become 123.00 and if it's changed to string it will become "123" which is not a number but a character in the form of a number.

```
ip: a=123
    type(a)
op: int

ip: a=str(a)
    type(a)
op: str

# And we can do this as vice versa to change strings(in the form of numbers) to float and int.
# And also float to int and str.

ip: int(123.45)
op: 123
```

Print Function:

print() is a built-in function in Python used to print desired output.

```
ip: print("asdfghjkl")
op: asdfghjkl

ip: a=12
    print(a)
op: 12

ip: print(1+2)
op: 3
```

Functionalities of Print() function:

- **\n** can be used to print text in separate lines.
- **sep** can be used to separate two output terms.

```
ip: print("asdfghjkl
    asdghhh)
op: error

ip: print("asdfghjkl\nasdghhh")
op: asdfghjkl
    asdfhhh

ip: a=12
    b="pen"
    print(a,b)
op: 12,pen

ip: print(a,b,sep='-')
op: 12-pen
```

Types of printing variables along with a text using Print():

- Using string concatenation (Adding the variable with text)
- Using comma
- fstring
- Formatting
- Placeholders using Format Function

Using String concatenation:

Just use the symbol '+' between two strings to add them.

```
ip: # Adding two variables in Print using concatenation
    # As this method adds only strings, we first need to typecast every variable as a string
    a=10
    b="harshith"
    print(str(a)+' '+b)
op: 10 harshith
```

Using comma:

Just use the comma symbol between two variables to add them(variables can be of any Data Types)

```
ip: # Add them using a comma
    print(a,b)
op: 10 harshith
```

fstring:

Add variables in between text in angular braces {} and use f at the point before the coats(",') of print statement.

```
ip: # let's think name=Harshith, age=21 and degree =Btech
    # print My name is Harshith and My age is 21 and I had done my BTech degree
    name="Harshith"
    age=21
    degree="Btech"
    print(f"My name is {name} and age is {age} and I had done {degree}")
op: My name is Harshith and age is 21 and I had done Btech
```

Format Function:

Same as fstring but there is no need to give f and also no need to give the variable name in angular {} braces. Instead, we can use .format() function and pass the variables or parameters in it.

```
ip: print("My name is {} and age is {} and I had done my {}".format(name,age,degree))
op: My name is Harshith and age is 21 and I had done Btech

# Note: The variables in the format function should be in the same order as the places
#       you want them to fit in angular braces.
```

Placeholders using Format Function:

we can use placeholders in angular braces so that there will be no need for particular order to be followed in format function.

```
ip: print("My name is {n} and age is {a} and I had done my {d}".format(a=age,d=degree
                                                                       ,n=name))
op: My name is Harshith and age is 21 and I had done my Btech
```

Control Statements in Python:

Python has several control statements like

If-Else: Executes a block of code based on whether a condition is true or false.

```
ip: a=22
    # using "IF" the code will check if the condition is greater than 18 or not.
    if a>18:
      # if the statement is true it will execute the below line of code
      print("You are major")
    else:
      # if the "IF" statement is false, then the Else code will be executed.
      print("You are minor")
op: You are major
```

Elif: Will be used along with If-Else where we have to check more than 2 conditions.

```
ip: a=77
    # using "IF" the code will check if the condition is greater than 18 or not.
    if a>18 and a<60:
      # if the statement is true it will execute the below line of code
      print("You are major")
    elif a>60:
      print("You are old")
    else:
      print("You are minor")
op: You are old
```

For Loop: Executes a block of code repeatedly for each item in a sequence (list, tuple, etc.).

For eg: if name="Harshith" and I want to traverse each of the elements of the value of the variable name, Then I have to travel through H A R S H I T H one by one. we can do this by using For loop. so we can define for-loop as the one which can traverse through all the elements of a given value be it string or list or dictionary or etc.

```
ip: name="HARSHITH"
    # Here i is the temporary variable to which we are assigning each element of the name
                                                          # "HARSHITH" one after another.
    for i in name:
      #Now we are going to print each value of i
      print(i)
op: H
    A
    R
    S
    H
    I
    T
    H
```

We can also use if-else statements in the for loop as below

```
#Program to replace all the letters "H" to "*" using if statement in for loop.
ip: name="HARSHITH"
    for i in name:
      # I will add an if statement here.
      if i=="H":
        print(*)
      else:
        print(i)
op: *
    A
    R
    S
    *
    I
    T
    *
```

Instead of Printing each character in a different line, we can print them on the same line using the end property in the print statement.

```
#In this code I used end="" empty string to be placed between each printing Element.
ip: name="HARSHITH"
    for i in name:
      if i=="H":
        print(*,end="")
      else:
        print(i,end="")
op: *ARS*IT*
```

Break: Terminates a loop prematurely.

we can use the Break statement to end a loop when a certain condition occurs.

```
#In this code I will stop executing code if there is letter "R" occurs.
ip: name="HARSHITH"
    for i in name:
      if i=="R":
        break
      print(i,end="")
op: HA
```

Continue: Skips the current iteration of a loop and moves to the next one.

we can use the Continue to skip the loop for one time to execute when a certain condition occurs.

```
#In this code I will skip executing code if there is letter "H" occurs.
ip: name="HARSHITH"
    for i in name:
      if i=="H":
        continue
      print(i,end="")
op: ARSIT
```

Pass: Does nothing, used as a placeholder for future code.

```
#In this code pass Does nothing, used as a placeholder for future code
ip: name="HARSHITH"
    for i in name:
      if i=="H":
        pass
      print(i,end="")
op: HARSHITH
```

While Loop: Executes a block of code repeatedly as long as a given condition is true.

For Eg: If there is a Government employee who aged 25. if we want to write a code to notify him when it's time for him to Retire we can do it using While loop by printing that it's time for his retirement once he reaches 62 by using While Loop.

```
ip: # While loop
    age=25
    experience=0
    # when age becomes 62 this while loop will stop
    # ! is not symbol. != this refer not equal to 62 in the below loop.
    while age!=62:
      # Here we are increasing the age by 1 each time while the loop is running.
      age=age+1
      #Let's increase experience every year
      experience=expericene+1
    else:
      print("it's time to Retire! You had now {} years of experience.format(experience))
op: it's time to retire you had now 37 years of experience.
```

Logical Operators:

There are 3 types of Logical Operators in Python, they are

- **And:** Return True if both the values are True, else return False.

Value 1	Value 2	Result of AND
True	True	True
True	False	False
False	True	False
False	False	False

- **Or :** Return True if any one value is True , else return False.

Value 1	Value 2	Result of OR
True	True	True
True	False	True
False	True	True
False	False	False

- **Not :** Return the opposite of the given value.

Value	Result of NOT
True	False
False	True

Comparision Operators:

The main Comparision Operators in python include:

- **is** - Returns True if both the a and b point to the same memory location, else Return False. Usually, non mutable Data Types with the same values in them return True.

```
# The below code will return False because lst1 and lst2 were saved in a different location.
ip: lst1=[1,2,3,4,5]
    lst2=[1,2,3,4,5]
    lst1 is lst2
op: False
```

```
# The below code will return True because a and b were saved in the same location.
ip: a=9
    b=9
    a is b
op: True
```

- **is not** **-** It is the opposite of the is operator

```
# The below code will return True because lst1 and lst2 were saved in different locations.
ip: lst1=[1,2,3,4,5]
    lst2=[1,2,3,4,5]
    lst1 is not lst2
op: True


# The below code will return False because a and b were saved in the same location.
ip: a=9
    b=9
    a is not b
op: False
```

- **==** **-** it will return True if the value assigned to **a** is equal to value assigned to **b**, else False.

```
# The below code will return True because lst1 and lst2 have the same values in them.
ip: lst1=[1,2,3,4,5]
    lst2=[1,2,3,4,5]
    lst1 == lst2
op: True


# The below code will return False because a and b have different values.
ip: a=9
    b=90
    a == b
op: False
```

- **! =** **-** It is exactly to the opposite of == Operator.

```
# The below code will return False because lst1 and lst2 have the same values in them.
ip: lst1=[1,2,3,4,5]
    lst2=[1,2,3,4,5]
    lst1 != lst2
op: False


# The below code will return True because a and b have different values.
ip: a=9
    b=90
    a != b
op: True
```

Arithmetic Operators:

- **+** - addition
- **-** - subtraction
- ***** - multiplication
- **/** - division
- **//** - division without decimal values
- **%** **-** remainder

Number 1	Number 2	Operator	Result
100	9	+	109
100	9	-	91
100	9	*	900
100	9	/	11.1111

100	9	//	11
100	9	%	1

Strings:

In Python, a string is a sequence of characters represented in quotes (single or double).

Slicing is a way to extract a portion of a string, list, or any sequence. It's done by specifying the start and end index of the portion, separated by a colon (:). For example:

```
ip: my_string = "Hello World!"
    my_string[0:5]
op: Hello
```

In this example, the slicing **my_string[0:5]** returns a new string that starts at index 0 and ends at index 5 (not including index 5).

If the start index is not specified, it defaults to the start of the sequence. If the end index is not specified, it defaults to the end of the sequence. For example:

```
ip: my_string[:5]
op: Hello
ip: my_string[6:]
op: World!
```

Negative indices can also be used to slice from the end of the sequence. For example:

```
ip: my_string[-6:]
op: World!
```

1. `len()`: returns the length of the string.
2. `find()`: returns the index of the first occurrence of the substring in the string. Returns -1 if the substring is not found.
3. `count()`: returns the number of occurrences of the substring in the string.
4. `partition()`: splits the string into three parts: the part before the first occurrence of the substring, the substring, and the part after the substring. Returns a tuple of these three parts.
5. `upper()`: returns a new string with all uppercase characters.
6. `lower()`: returns a new string with all lowercase characters.
7. `swapcase()`: returns a new string with all uppercase characters converted to lowercase and all lowercase characters converted to uppercase.
8. `title()`: returns a new string with the first letter of each word capitalized.
9. `capitalize()`: Capitalizes the first character of a string.
10. `reversed()`: Returns a reverse iterator of a string. To get the reversed string, use `''.join(reversed(string))`.
11. `strip()`: Removes leading and trailing whitespaces from a string.
12. `lstrip()`: Removes leading whitespaces from a string.
13. `rstrip()`: Removes trailing whitespaces from a string.
14. `replace()`: Replaces a substring in a string with another string.
15. `isspace()`: Returns `True` if all characters in a string are whitespaces, `False` otherwise.
16. `isupper()`: Returns `True` if all characters in a string are uppercase, `False` otherwise.
17. `islower()`: Returns `True` if all characters in a string are lowercase, `False` otherwise.
18. `startswith()`: Returns `True` if a string starts with a specified prefix, `False` otherwise.
19. `endswith()`: Returns `True` if a string ends with a specified suffix, `False` otherwise.
20. `isalnum()`: Returns `True` if all characters in a string are alphanumeric (letters and digits), `False` otherwise.

21. `isdigit()`: Returns `True` if all characters in a string are digits, `False` otherwise.
22. `isalpha()`: Returns `True` if all characters in a string are letters, `False` otherwise.

Usage of String Functions:

```
string = "    Hello World!  "

print("Length of the string: ", len(string))

substring = "Hello"
print("First occurrence of substring '{}' at index: {}".format(substring, string.find(substring)))
print("Number of occurrences of substring '{}': {}".format(substring, string.count(substring)))

before, sep, after = string.partition("World")
print("Partition of 3 blocks",before,sep,after)

print("Uppercase: ", string.upper())
print("Lowercase: ", string.lower())
print("Swapcase: ", string.swapcase())
print("Title: ", string.title())
print("Capitalize: ", string.capitalize())

print("Reversed: ", ''.join(reversed(string)))

print("Strip: ", string.strip())
print("Left Strip: ", string.lstrip())
print("Right Strip: ", string.rstrip())

replacement_string = string.replace("Hello", "Hi")
print("Replaced 'Hello' with 'Hi': ", replacement_string)

print("Is all whitespace? ", string.isspace())
print("Is all uppercase? ", string.isupper())
print("Is all lowercase? ", string.islower())
print("Starts with 'Hello'? ", string.startswith("Hello"))
print("Ends with '!'? ", string.endswith("!"))
print("Is alphanumeric? ", string.isalnum())
print("Is digit? ", string.isdigit())
print("Is alphabet? ", string.isalpha())
```

Outputs:

```
Length of the string:  17
First occurrence of substring 'Hello' at index: 5
Number of occurrences of substring 'Hello': 1
Partition of string: '    Hello ' | 'World' | '!  '
Uppercase:  HELLO WORLD!
Lowercase:  hello world!
Swapcase:  HELLO wORLD!
Title:  Hello World!
Capitalize:  Hello world!
Reversed: !  dlroW olleH
Strip: Hello World!
Left Strip: Hello World!
Right Strip:     Hello World!
Replaced 'Hello' with 'Hi':     Hi World!
Is all whitespace?  False
Is all uppercase?  False
Is all lowercase?  False
Starts with 'Hello'?  False
Ends with '!'?  True
Is alphanumeric?  False
Is digit?  False
Is alphabet?  False
```

Lists:

A list is a collection of items in a ordered sequence, which can be of different data types (integer, string, float, etc.). Lists are defined using square brackets `[]` and items are separated by commas `,`.

- `min` is a built-in function in Python that returns the minimum value from a list or any iterable.
- `max` is a built-in function in Python that returns the maximum value from a list or any iterable.

- `sort` is a method in Python that can be used to sort a list in ascending or descending order. By default, the `sort` method sorts the list in ascending order.
- `reverse` is a method in Python that can be used to reverse the order of elements in a list.

  descending reverse: To sort a list in descending order, the `sort` method can be used with the argument `reverse=True`.
- `append` is a method in Python that can be used to add an element to the end of a list.
- `extend` is a method in Python that can be used to add elements from one list to another.

```
# Initializing a list
numbers = [3, 4, 1, 8, 5]

# Using the min function
min_value = min(numbers)
print("Min Value:", min_value)

# Using the max function
max_value = max(numbers)
print("Max Value:", max_value)

# Using the sort method
numbers.sort()
print("Sorted List:", numbers)

# Using the reverse method
numbers.reverse()
print("Reversed List:", numbers)

# Using the sort method with reverse argument
numbers.sort(reverse=True)
print("Sorted List in Descending Order:", numbers)

# Using the append method
numbers.append(9)
print("List after appending 9:", numbers)

# Using the extend method
new_numbers = [10, 11, 12]
numbers.extend(new_numbers)
print("List after extending with new numbers:", numbers)
```

Outputs:

```
Min Value: 1
Max Value: 8
Sorted List: [1, 3, 4, 5, 8]
Reversed List: [8, 5, 4, 3, 1]
Sorted List in Descending Order: [8, 5, 4, 3, 1]
List after appending 9: [8, 5, 4, 3, 1, 9]
List after extending with new numbers: [8, 5, 4, 3, 1, 9, 10, 11, 12]
```

List Comprehensions:

List comprehensions are a concise way to create new lists by transforming existing ones. They are written as a single line of code and can include an optional condition.

```
# Using List Comprehensions
numbers=[8, 5, 4, 3, 1, 9, 10, 11, 12]
squared_numbers = [x**2 for x in numbers]
print("Squared Numbers:", squared_numbers)
```

Output:

```
Squared Numbers: [64, 25, 16, 9, 1, 81, 100, 121, 144]
```

Tuples:

- Tuples are immutable, ordered collections of items, denoted by ()

- Each item in a tuple is called an element and is separated by a comma.

Functions in Tuple:

1. **index(element):** returns the index of the first occurrence of the given element in the tuple.
2. **count(element):** returns the number of occurrences of the given element in the tuple.
3. **len(tuple):** returns the number of elements in the tuple.
4. **min(tuple):** returns the smallest element in the tuple.
5. **max(tuple):** returns the largest element in the tuple.

```
# Creating a tuple
t = (1, 2, 3, 3, 4, 5)

# Index function
print("The index of 3 in the tuple is: ", t.index(3))
# Output: The index of 3 in the tuple is: 2

# Count function
print("The count of 3 in the tuple is: ", t.count(3))
# Output: The count of 3 in the tuple is: 2

# Length function
print("The length of the tuple is: ", len(t))
# Output: The length of the tuple is: 6

# Min function
print("The minimum element in the tuple is: ", min(t))
# Output: The minimum element in the tuple is: 1

# Max function
print("The maximum element in the tuple is: ", max(t))
# Output: The maximum element in the tuple is: 5

# Concatenation of two tuples
t1 = (6, 7)
t2 = t + t1
print("The concatenated tuple is: ", t2)
# Output: The concatenated tuple is: (1, 2, 3, 3, 4, 5, 6, 7)

# Changing elements (Convert tuple to list, change elements and convert list to tuple)
l = list(t2)
l[0] = 8
t2 = tuple(l)
print("The modified tuple is: ", t2)
# Output: The modified tuple is: (8, 2, 3, 3, 4, 5, 6, 7)
```

Adding items to a tuple:

- Tuples are immutable, meaning once you create a tuple you cannot change its elements.
- However, you can concatenate two tuples to form a new tuple, e.g. tuple1 + tuple2.

Changing elements:

- Tuples are immutable, meaning you cannot change its elements.
- If you need to change elements in a tuple, you'll have to convert the tuple to a list, change the elements and then convert the list back to a tuple.

The **del** statement is used in Python to delete a reference to an object. When you delete a reference, the object may not be immediately destroyed. Instead, the memory occupied by the object is released when there are no more references to the object.

```
t = (1, 2, 3, 4)
print("The original tuple: ", t)
# Output: The original tuple: (1, 2, 3, 4)

# Deleting the tuple
del t

# Accessing the tuple after deletion
# This will raise an error because the tuple has been deleted
print("The deleted tuple: ", t)
# Output: NameError: name 't' is not defined
```

Sets:

A set in Python is an unordered collection of unique elements.

Functions:

1. `add()`: Adds an element to the set.
2. `pop()`: Removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.
3. `clear()`: Removes all elements from the set.

```
# Creating a set
s = {1, 2, 3}
print("The set: ", s)
# Output: The set: {1, 2, 3}

s = {1, 2, 3}
print("The original set: ", s)
# Output: The original set: {1, 2, 3}

s.add(4)
print("The set after adding an element: ", s)
# Output: The set after adding an element: {1, 2, 3, 4}

s = {1, 2, 3}
print("The original set: ", s)
# Output: The original set: {1, 2, 3}

x = s.pop()
print("The popped element: ", x)
# Output: The popped element: 1
print("The set after popping an element: ", s)
# Output: The set after popping an element: {2, 3}

s = {1, 2, 3}
print("The original set: ", s)
# Output: The original set: {1, 2, 3}

s.clear()
print("The set after clearing all elements: ", s)
# Output: The set after clearing all elements: set()
```

Adding elements to a set:

Sets can only contain immutable elements. Therefore, lists can't be added to a set directly, but tuple elements can be added to the set.

```
s = {1, 2, 3}
print("The original set: ", s)
# Output: The original set: {1, 2, 3}

# Adding a tuple to the set
s.add((4, 5))
print("The set after adding a tuple: ", s)
# Output: The set after adding a tuple: {1, 2, 3, (4, 5)}
```

Python Dictionaries

A dictionary in Python is an unordered collection of key-value pairs. It is also known as an associative array or hash map. You can use a dictionary to store values in a key-value format, where each key maps to a unique value.

```
# Creating a dictionary
d = {'key1': 'value1', 'key2': 'value2'}
print("The dictionary: ", d)
# Output: The dictionary: {'key1': 'value1', 'key2': 'value2'}
```

Dictionary Methods

Here are some of the common methods that can be performed on a dictionary:

1. `keys()`: Returns a view object that displays a list of all the keys in the dictionary.
2. `values()`: Returns a view object that displays a list of all the values in the dictionary.
3. `items()`: Returns a view object that displays a list of all the key-value pairs in the dictionary.
4. `update()`: Adds elements to the dictionary. If the key is already present, it updates the key-value.
5. The `update()` method is used to add elements to an existing dictionary or to merge two dictionaries.
6. The `pop()` method is used to remove an item from the dictionary.
7. The `get()` method is used to retrieve the value of a key from the dictionary.
8. The `fromkeys()` method is used to create a new dictionary where each key is the same and the values are set to None.
9. To create a deep copy of a dictionary, you can use the `copy` module in Python.

```
# Creating a dictionary
d = {'key1': 'value1', 'key2': 'value2'}
print("The dictionary: ", d)
# Output: The dictionary: {'key1': 'value1', 'key2': 'value2'}

d = {'key1': 'value1', 'key2': 'value2'}
print("The keys of the dictionary: ", d.keys())
# Output: The keys of the dictionary: dict_keys(['key1', 'key2'])

d = {'key1': 'value1', 'key2': 'value2'}
print("The values of the dictionary: ", d.values())
# Output: The values of the dictionary: dict_values(['value1', 'value2'])

d = {'key1': 'value1', 'key2': 'value2'}
print("The items of the dictionary: ", d.items())
# Output: The items of the dictionary: dict_items([('key1', 'value1'), ('key2', 'value2')])

d = {'key1': 'value1', 'key2': 'value2'}
print("The original dictionary: ", d)
# Output: The original dictionary: {'key1': 'value1', 'key2': 'value2'}

d.update({'key3': 'value3'})
print("The dictionary after updating an element: ", d)
# Output: The dictionary after updating an element: {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

import copy

d = {'key1': 'value1', 'key2': 'value2'}
print("The original dictionary: ", d)
# Output: The original dictionary: {'key1': 'value1', 'key2': 'value2'}
d_copy = copy.deepcopy(d)
print("The deep copy of the dictionary: ", d_copy)
# Output: The deep copy of the dictionary: {'key1': 'value1', 'key2': 'value2'}

dict = {'key1': 'value1', 'key2': 'value2'}
dict2 = {'key3': 'value3', 'key4': 'value4'}
dict.update(dict2)
print("The updated dictionary: ", dict)
# Output: The updated dictionary: {'key1': 'value1', 'key2': 'value2', 'key3': 'value3', 'key4': 'value4'}

dict = {'key1': 'value1', 'key2': 'value2'}
dict.pop('key1')
print("The dictionary after popping key1: ", dict)
# Output: The dictionary after popping key1: {'key2': 'value2'}

dict = {'key1': 'value1', 'key2': 'value2'}
print("The value of key1: ", dict.get('key1'))
# Output: The value of key1: value1

keys = ['key1', 'key2', 'key3']
dict = dict.fromkeys(keys, 'value')
print("The dictionary created from keys: ", dict) # Output: The dictionary created from keys: {'key1': 'value', 'key2': 'value', 'key3
```

Functions:

- Functions are blocks of reusable code that can be called multiple times in a program.
- They are defined using the `def` keyword, followed by the function name, parameters within parentheses and a colon.
- The code inside the function is indented and executed when the function is called.
- Functions can return values using the `return` keyword. If a function does not have a return statement, it returns `None` by default.

```
def greet(name):
    """This function greets the person passed in as a parameter"""
    print("Hello, " + name + ". How are you today?")

greet("John")
# Output: Hello, John. How are you today?
```

Arguments:

- Functions can take arguments as inputs.
- There are different types of arguments:
  - Positional arguments: passed to the function in the same order as they are defined in the function.
  - Keyword arguments: passed to the function using the argument name and value.
  - Default arguments: assigned a default value in the function definition. If no value is provided during the function call, the default value is used.
  - Variable-length arguments: the number of arguments can be varied.

```
def greet(name, msg="Good morning!"):
    """This function greets the person passed in as a parameter"""
    print("Hello, " + name + ". " + msg)

greet("John", "How do you do?")
# Output: Hello, John. How do you do?

greet("John")
# Output: Hello, John. Good morning!
```

*args:

- It allows the function to take a variable number of positional arguments.
- `args` is used as a parameter in the function definition and collects all remaining positional arguments into a tuple.

```
def greet(*names):
    """This function greets all persons in the names tuple"""
    for name in names:
        print("Hello, " + name)

greet("John", "Jane", "Jim")
# Output:
# Hello, John
# Hello, Jane
# Hello, Jim
```

**kwargs:

- It allows the function to take a variable number of keyword arguments.
- `*kwargs` is used as a parameter in the function definition and collects all remaining keyword arguments into a dictionary.

```
def greet(**kwargs):
    """This function greets the person passed in as a parameter"""
    if kwargs:
        print("Hello, " + kwargs['name'] + ". " + kwargs['msg'])

greet(name="John", msg="How do you do?")
# Output: Hello, John. How do you do?
```

Generator Functions:

- A generator function is a special type of function that can be used to generate a sequence of values over time, instead of computing all the values at once and returning them in a list.
- The generator function uses the `yield` keyword instead of `return` to produce a value.

- Each time the generator function is called, it resumes execution from where it was paused and runs until it encounters the next `yield` statement.
- Generator functions are used for iteration, because they can generate values one by one, instead of all at once.

```
def my_range(n):
    """This is a generator function that yields values from 0 to n (exclusive)"""
    i = 0
    while i < n:
        yield i
        i += 1

for i in my_range(5):
    print(i)
# Output:
# 0
# 1
# 2
# 3
# 4
```

Lambda Function:

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

```
x = lambda a : a + 10
print(x(5))
# Output: 15
```

Map:

The `map()` function is used to apply a function to each item of an iterable (list, tuple etc.) and returns a new list containing all the items modified by the function

```
pythonCopy code
def multiply(x):
  return x * 2

numbers = [1, 2, 3, 4]
result = list(map(multiply, numbers))
print(result)
# Output: [2, 4, 6, 8]
```

Reduce:

The `reduce()` function is used to apply a particular function passed in its argument to all of the list elements. This function is defined in the `functools` module.

```
pythonCopy code
from functools import reduce

def multiply(x, y):
  return x * y

numbers = [1, 2, 3, 4]
result = reduce(multiply, numbers)
print(result)
# Output: 24
```

Filter:

The `filter()` function is used to filter the given iterable with the help of another function passed as an argument to test all the elements to be `True` or `False`.

```
def even_check(num):
  if num % 2 == 0:
    return True
```

```
numbers = [1, 2, 3, 4, 5, 6]
result = list(filter(even_check, numbers))
print(result)
# Output: [2, 4, 6]
```

OOPS in Python:

Object:

An object is an instance of a class, which represents real-world entities such as a person, a car, a bank account, etc. Everything in Python is an object, including data types such as numbers, strings, and lists.

```
x = 10 # 10 is an object of int class
y = "hello" # "hello" is an object of str class
z = [1,2,3] # [1,2,3] is an object of list class
```

Class:

A class is a blueprint for creating objects, allowing you to define the attributes and behaviors of objects of a particular type.

```
class Car:
   def __init__(self, make, model, year):
      self.make = make
      self.model = model
      self.year = year
```

Method:

A method is a function that is associated with an object and can be called on that object.

```
class Car:
   def __init__(self, make, model, year):
      self.make = make
      self.model = model
      self.year = year
   def get_info(self):
      return f"{self.make} {self.model} ({self.year})"
```

Constructor:

The `__init__` method is a special method that is called when an object is created from a class and is used to initialize the attributes of the object. It is also called a constructor.

```
class Car:
   def __init__(self, make, model, year):
      self.make = make
      self.model = model
      self.year = year

car = Car("Toyota", "Camry", 2020)
print(car.make) # Output: Toyota
print(car.model) # Output: Camry
print(car.year) # Output: 2020
```