String

1. **capitalize() -** convert the first character of a string to uppercase and the rest of the characters to lowercase
    1. **string.capitalize()**
2. **title() -** Returns a new string with the first character of each word in the original string capitalized (made uppercase), while the rest of the characters are in lowercase. Words are recognized as sequences of characters separated by whitespace.
    1. **string.title()**
3. **lower() - string.lower()**
4. **upper() - string.upper()**
5. **islower() - string.islower() -** Returns “True or False”.
6. **isupper() - string.isupper() -** Returns “True or False”.
7. **Swapcase() -** Returns a new string with the case of each letter in the original string swapped. That is, it converts uppercase letters to lowercase and vice versa.
    1. **string.swapcase()**
8. **format()**
    1. "My name is {} and I am {} years old.".format(name, age)
    2. "My name is {0} and I am {1} years old. Let's say it again: {1}, {0}.".format(name, age)
    3. f"My name is {name} and I am {age} years old."
        - name = "Alice"
        - age = 30
9. **format_map()**
    1. "My name is {name}, I am {age} years old, and I live in {city}.".format_map(data)
        - data = {'name': 'David', 'age': 28, 'city': 'New York'}
10. **count() -** Returns the number of times that substring appears
    1. **string.count(substring[, start[, end]])**
11. **replace() - string.replace(old, new[, count])**
12. **find () -** Returns “index of the first occurrence” If the substring is not found, it returns -1.
    1. **string.find("substring [, start[, end]])**
13. **index () -** However, unlike **find()**, if the substring is not found, index() raises a ValueError
    1. **string.index("substring ")**
14. **rfind()- string.rfind(substring[, start[, end]])**
    1. Returns “index of the first occurrence” If the substring is not found, it returns -1.
    2. The **rfind()** method is similar to **find()**, but it starts the search from the end of the string, moving towards the beginning.
15. **rindex()- string.rindex(substring[, start[, end]])**
    1. However, unlike **rfind()**, if the substring is not found, index() raises a ValueError
16. **Join() –** joins elements of an iterable (such as a list, tuple, or string) into a single string.
    1. " - ".join(map(str, numbers))
    2. delimiter.join(words)
17. **Split() - string.split([separator[, maxsplit]])**
18. **rsplit() –** split a string into a list of substrings based on a specified separator, starting from the right (end) of the string. It is similar to the **split()** method but works in the opposite direction.
    1. **string.rsplit([separator[, maxsplit]])**.
19. **splitlines() - string.splitlines(keepends=True)**
    1. keepends (optional): An optional boolean parameter. If set to True, the line breaks are included in the resulting lines. If set to False or omitted, the line breaks are excluded.
20. **partition()-** split a string into three parts based on the first occurrence of a specified separator
    1. **string.partition(separator)**

21. **rpartition() -** splits a string into three parts based on the last occurrence of a specified separator.
    1. **string.rpartition(separator)**
22. **endswith () - string.endswith(suffixes)** – Returns "True or False"
23. **Startswith() - string.startswith(prefix[, start[, end]])**
    1. prefix: The substring to check for at the beginning of the string.
    2. start (optional): The starting index for the check. If specified, the check starts from this index. Default is 0.
    3. end (optional): The ending index for the check. If specified, the check goes up to (but does not include) this index. Default is the length of the string.
24. **removeprefix() - string.removeprefix(prefix)**
25. **removesuffix() - string.removesuffix(suffix)**
26. **isalnum() - string1.isalnum()** - Returns "True or False". Alphanumeric characters are those that are either letters (a-z, A-Z) or numbers (0-9)
27. **isalpha() - string1.isalpha()** - Returns "True or False". Alphanumeric characters are letters (a-z, A-Z)
28. **isascii() - string1.isascii()** - Returns "True or False". used to check whether all characters in a string are ASCII (American Standard Code for Information Interchange) characters
29. **isdecimal() - string1.isdecimal()** - Returns **True** if all characters in the string are decimal digits **(0-9)** and the string is not empty. Otherwise, it returns **False**. It does not consider negative numbers, floating-point numbers, or other non-digit characters besides 0-9.
30. **isdigit() - string1.isdigit()** - Returns "True or False". However, unlike **isdecimal()**, Digits include decimal digits **(0-9)** and other digit characters, such as those used in other numeral systems **(I,II,III,IV,V…,IX,X….).**
31. **isidentifier() - string. isidentifier() -** Returns "True or False".
    1. It must start with a letter (a-z, A-Z) or an underscore (_).
    2. The rest of the characters can be letters, digits (0-9), or underscores.
32. **isnumeric() - string1.isnumeric() -** Returns "True or False". However, unlike **isdigit()**, including digits, fractions**(½)**, superscripts**(²³)**, subscripts**(₀₁)**, and other numeric characters that are not part of the decimal system**(I,II,III,IV,V…,IX,X….)**.
33. **isprintable() - string.isprintable() -** Returns "True or False". A character is considered printable if it can be rendered on the screen and is not a control character **(\n)**.
34. **isspace() - string.isspace() -** Returns "True or False". Whitespace characters include spaces **( )**, tabs **(\t)**, and newline **(\n)** characters. If all characters in the string are whitespace, the method returns **True**; otherwise, it returns **False**.
35. **istitle() - string.istitle() -** Returns "True or False".
    1. The first character of each word is in uppercase (title case).
    2. All remaining characters in each word are in lowercase.
36. **ljust() -** returns a left-justified version of the string. It pads the original string with a specified character (default is space) to a specified width, ensuring that the original content is left-aligned within the resulting string.
    1. **string.ljust(width, fillchar)**
        - **width**: The total width of the resulting string, including the original content and any padding.
        - **fillchar**: (Optional) The character used for padding. If not specified, space character is used by default.
37. **rjust() -** returns a right-justified version of the string.
    1. **string.rjust(width, fillchar)**
        - **width**: The total width of the resulting string, including the original content and any padding.

- **fillchar**: (Optional) The character used for padding. If not specified, space character is used by default.

38. **center () - string.center(width, fill_character)**
39. **zfill() -** pads a numeric string with zeros (0) on the left side to achieve a specified width.
    1. **string.zfill(width)**
40. **lstrip() =** Returns a new string with leading (leftmost) whitespace characters removed.
    1. **string.lstrip([characters])**
        - **characters** (optional): A string specifying the set of characters to be removed. If not provided, it removes leading whitespace characters (spaces, tabs, and newlines) by default.
41. **rstrip() -** Returns a new string with trailing (rightmost) whitespace characters removed.
    1. **string.rstrip([characters])**
        - **characters** (optional): A string specifying the set of characters to be removed. If not provided, it removes trailing whitespace characters (spaces, tabs, and newlines) by default
42. **Strip() -** Returns a new string with leading (leftmost) and trailing (rightmost) whitespace characters removed. It does not modify the original string; instead, it creates a new string with the specified characters removed from both ends of the original string.
    1. **string.strip([characters])**
        - characters (optional): A string specifying the set of characters to be removed. If not provided, it removes leading and trailing whitespace characters (spaces, tabs, and newlines) by default.
43. **maketrans()**
    1. **str.maketrans(x[, y[, z]])**
        - **x**: A string specifying the characters to be replaced.
        - **y**: A string specifying the characters to replace **x** with.
        - **z**: A string specifying characters to be deleted.
    2. **str.maketrans("aeiou", "12345")**
    3. **str.maketrans("", "", "aeiou")**
44. **translate() -** Returns a new string where some specified characters are replaced with the character described in a dictionary or mapping table. It is a powerful method that allows you to perform complex character-based replacements in a string.
    1. **string.translate(str.maketrans("eo", "36"))**
    2. **string.translate(translation_table)**
        - translation_table = str.maketrans("eo", "36")
45. **string.casefold()**
46. **string.encode('ascii', errors='ignore') -** Returns an encoded version of the string
47. **string.expandtabs(number of spaces)** – Result: Hello world example

---

List = [element1, element2, element3,..]

1. **append() -** to add an element to the end of a list.
    - **list_name.append(element)**
2. **insert() -** is used to insert an element at a specific index in a list.
    - **list_name.insert(index, element)**
3. **index() -** is used to find the index of the first occurrence of a specified element in a list.
    - **index = list_name.index(element, start, end)**
4. **count() -** is used to count the number of occurrences of a specified element in a list.

- count = list_name.count(element)

5. **copy() -** is used to create a shallow copy of a list
    - **new_list = old_list.copy()**
6. **extend() -** to extend a list by appending elements from an iterable (e.g., another list, tuple, or any iterable)
    - **list_name.extend(iterable)**
        - **list1 = [1, 2, 3]**
        - **list2 = [4, 5, 6]**
        - **list1.extend(list2)**
7. **pop() -** is used to remove and return the element at a specified index from a list. If the index is not specified, the method will remove and return the last element:
    - **removed_element = list_name.pop(index)**
8. **remove() -** to remove the first occurrence of a specified element from a list.
    - **list_name.remove(element)**
9. **clear() -** to remove all elements from a list.
    - **list_name.clear()**
10. **reverse()** - to reverse the elements of a list in place.
    - **list_name.reverse()**
    - **reversed_list = my_list[::-1]**
11. **sort()** - to sort the elements of a list in ascending order.
    - **list_name.sort(key=None, reverse=False)**
        - **key**: A function to extract a comparison key from each element. If not provided, the elements are sorted based on their natural order.
        - **reverse**: A boolean indicating whether to sort the list in descending order. The default is False, which sorts the list in ascending order.
    - **my_list.sort()**
    - **my_list.sort(reverse=True)**
    - **my_list.sort(key=len) -** Length of Strings
    - **my_list.sort(key=abs) -** Absolute Values
    - **my_list.sort(key=lambda x: x[1]) -** Custom Function
    - **my_list.sort(key=lambda x: x.lower()) -** Case-Insensitive Sorting

---

Tuple = (element1, element2, element3,..)

1. **count()-** to count the number of occurrences of a specified element in a tuple.
    - **tuple.count(element)**
2. **index() -** to find the index of the first occurrence of a specified element in a tuple.
    - **tuple.index(element, start, end)**

Dictionary = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

1. **items()** method is used to return a view of the dictionary's key-value pairs as tuples.
    - **dictionary_items = my_dict.items()**
2. **keys() -** to get a view object that displays a list of all the keys in a dictionary.
    - **dictionary_keys = my_dict.keys()**
3. **values()** – to get a view object that displays a list of all values in a dictionary.
    - **values_view = my_dict.values()**

4. **get() -** to retrieve the value associated with a given key in a dictionary. It takes two parameters: the key whose value you want to retrieve and an optional default value to be returned if the key is not found in the dictionary.
    - **value = dictionary.get(key, default_value)**
        - **key**: The key whose value you want to retrieve.
        - **default_value** (optional): The value to be returned if the key is not found. If not provided, the default value is None.
5. **pop()** – to remove and return an item from a dictionary based on the specified key.
    - **value = my_dict.pop(key, default_value)**
        - **key:** The key of the item to be removed.
        - **default_value** (optional): The value to be returned if the key is not found. If not provided and the key is not found, a KeyError will be raised.
6. **update()** - to update a dictionary with key-value pairs from another dictionary, or from an iterable of key-value pairs (such as a list of tuples). If a key from the second dictionary already exists in the first dictionary, the corresponding value in the first dictionary is updated.
    - **my_dict.update(iterable_or_dictionary)**
    - **dict1.update(dict2)**
        - dict1 = {'a': 1, 'b': 2}
        - dict2 = {'b': 3, 'c': 4}
    - **my_dict.update([('b', 3), ('c', 4)])**
7. **popitem()** -
8. **clear()** - to remove all items from a dictionary. It doesn't take any parameters and returns **None**. After calling **clear()**, the dictionary becomes empty.
    - **my_dict.clear()**
9. **copy() -** to create a shallow copy of a dictionary. If the items in the original dictionary are mutable (e.g., lists or other dictionaries), changes to those mutable items will be reflected in both the original and the copied dictionaries.
    - **copied_dict = original_dict.copy()**
10. **fromkeys()** - returns a new dictionary with keys from a given iterable (e.g., a list, tuple, or range) and all values set to a specified default value or **None** if no default value is provided.
    - new_dict = dict.fromkeys(iterable, default_value)
        - iterable: The iterable whose elements will become the keys of the new dictionary.
        - default_value (optional): The value to be assigned to each key in the new dictionary. If not provided, the default value is None.
11. **setdefault()** -

---

lambda

short, simple operations where a full function definition might be overkill. They are particularly useful in situations where you need to pass a small function as an argument to higher-order functions like **map()**, **filter()**, or **sorted()**.

Functions name = lambda arguments: expression

- **Functions name (arguments)**

Functions name = lambda arguments1, arguments2, arguments3,… : expression

- **Functions name (arguments1, arguments2, arguments3,..)**

Example:

- key=lambda x: x.islower()
    - print(key("Giri"))
- lambda_square = lambda x: x ** 2
    - print(square(4))
- add = lambda x, y, z : x + y + z
    - print(add(3, 4, 5))
- squared_numbers = **map**(lambda x: x**2, numbers)
    - numbers = [1, 2, 3, 4, 5]
- sum_result = **map**(lambda x, y: x + y, numbers1, numbers2)
    - numbers1 = [1, 2, 3]
    - numbers2 = [4, 5, 6]
- sorted_data = **sorted**(data, key=lambda y: (y[1], y[2]))
    - data = [(1, 5, 3), (2, 3, 1), (3, 2, 4)]
- filtered_numbers = filter(lambda x: x % 2 == 0, numbers)
    - numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Example 1

```
import re
find_email = lambda text: re.search(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)

text = "Please contact gdharan1005@gmail.com for assistance."
result = find_email(text)
if result:
   print("Email found:", result.group())
else:
   print("No email found.")
```

---

if

Comparison Operators:

1. ==: Equal to
2. !=: Not equal to
3. <: Less than
4. >: Greater than
5. <=: Less than or equal to
6. >=: Greater than or equal to

syntax:

1. **if x == 10:**

Logical Operators:

1. and: Logical AND

2. or: Logical OR
3. not: Logical NOT

syntax:

1. **if x > 0 and y > 0:**
2. **if (x > 0 and y > 0) or z == 10:**

Membership Operators:

1. in: True if a value is found in the sequence
2. not in: True if a value is not found in the sequence

syntax:

1. **if "apple" in fruits:**
   - **fruits = ["apple", "banana", "orange"]**

Identity Operators:

1. is: True if both variables point to the same object
2. is not: True if both variables do not point to the same object

syntax:

1. **if list1 is list2:**
   - list1 = [1, 2, 3]
   - list2 = [1, 2, 3]

Truth Value Testing:

Any value that evaluates to True or False can be used directly in an if statement.

syntax:

1. **if my_list:**
   - my_list = [1, 2, 3] # my_list is not empty, so it evaluates to True
   - my_string = "Hello, World!" #The string is not empty, so it evaluates to True
   - my_dict = {"name": "John", "age": 25} #The dictionary is not empty, so it evaluates to True
2. **if x:**
   - x = 42 #x is not zero, so it evaluates to True

---

if – syntax

Example 1:

```
num = 7

if num > 0:
```

```
print(f"{num} is a positive number.")
```

Single-line if else statement:

- num = 7
  if num > 0: print(f"{num} is a positive number.")

- num = 7
  print(f"{num} is a positive number.") if num > 0 else None
  - **Note:** need to include an else clause. In Python, the ternary conditional expression (one-liner) requires both the true and false branches.

Example 2:

```
user_input = input("Enter a color: ")

if user_input.lower() == "blue":
   print("You chose the color blue!")
```

---

if else - syntax

Example 1:

```
number = int(input())

if number % 2 == 0:
   print(f"{number} is even.")
else:
   print(f"{number} is odd.")
```

Single-line if else statement:

```
y = 7
result = "Even" if y % 2 == 0 else "Odd"
print(result)
```

Example 2:

```
age = int(input())

if age >= 18:
   print("You are eligible to vote.")
else:
   print("You are not eligible to vote yet.")
```

---

elif - syntax

Example 1:

```
score = int(input())

if score >= 90:
   print("A")
elif score >= 80:
   print("B")
elif score >= 70:
   print("C")
elif score >= 60:
   print("D")
else:
   print("F")
```

Single-line elif statement:

```
z = 15
grade = "A" if z > 90 else "B" if z > 80 else "C" if z > 70 else "D" if z > 60 else "F"
print(grade)
```

Example 2:

```
hour = 15

if 6 <= hour < 12:
   print("Good morning!")
elif 12 <= hour < 18:
   print("Good afternoon!")
elif 18 <= hour < 24:
   print("Good evening!")
else:
   print("Good night!")
```

Nested if - syntax

Example 1:

```
number = -5

if number >= 0:
   if number == 0:
      print("The number is zero.")
   else:
      print("The number is positive.")
else:
   print("The number is negative.")
```

Single-line Nested if statement:

1. number = -5

```
print("The number is zero." if number == 0 else "The number is positive." if number >= 0 else "The number is negative.")
```

2.
```
num = 42
message = "Even and positive" if num > 0 and num % 2 == 0 else "Odd or non-positive"
print(message)
```

Example 2:

```
salary = int(input("Salary: "))
age = int(input("Age: "))

if salary >= 20000 or age <= 25:
    loan = int(input("Loan amount: "))

    if loan > 50000:
        print("Maximum loan amount is 50000")
    else:
        print("You are eligible for a loan")
else:
    print("You are not eligible for a loan")
```

Single-line If condition

Example 1:

```
my_list = [10, 20, 30]
first, second = my_list[0] if len(my_list) > 0 else 0, my_list[1] if len(my_list) > 1 else 0
print(first, second)
```

Example 2:

```
my_dict = {'name': 'Alice', 'age': 25}
name = my_dict.get('name') if 'name' in my_dict else 'unknown'
age = my_dict.get('age') if 'age' in my_dict else 'unknown'
print(name, age)
```

Example 3:

```
x, y, z = 10, 20, 30
result1, result2 = "OK" if x > 0 and y > 0 else "Not OK", "Valid" if z < 50 else "Invalid"
print(result1, result2)
```

```
y[1] if len(y) > 1 else 0, y[2] if len(y) > 2 else 0)
```

---

while loop

- **while condition:**

```
# code to be executed while the condition is true
# this block of code will keep executing as long as the condition is true
```

Some example for while loop syntax

Counting Down:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

User Input Validation:

```
valid_input = False
while not valid_input:
    user_input = input("Enter a positive number: ")
    if user_input.isdigit() and int(user_input) > 0:
        valid_input = True
    else:
        print("Invalid input. Please enter a positive number.")
```

Menu Driven Program:

```
choice = None
while choice != 'q':
    print("1. Option 1")
    print("2. Option 2")
    print("q. Quit")
    choice = input("Enter your choice: ")
    if choice == '1':
        print("Executing Option 1")
    elif choice == '2':
        print("Executing Option 2")
    elif choice == 'q':
        print("Quitting the program")
    else:
        print("Invalid choice. Please try again.")
```

Using a Break Statement:

```
correct_password = "secret"

while True:
    user_input = input("Enter the password: ")

    if user_input == correct_password:
```

```
        print("Access granted. Welcome!")
        break  # Exit the loop when the correct password is entered
    else:
        print("Incorrect password. Try again.")
```

---

for loop

for loop is used to iterate over a sequence (such as a list, tuple, string, or range) or other iterable objects

- for variable in iterable:
  # Code to be executed for each iteration

Types of Loop Variables:

1. Single Value Loop Variable:
   - for element in iterable:
     #element (or) i (or) j (or) name
2. Multiple Iterables:
   - for x, y in coordinates:
     - coordinates = [(1, 2), (3, 4), (5, 6)]
   - for key, value in my_dict.items():
3. Index and Value (enumerate):
   - for index, value in enumerate(iterable):
     # 'index' is the index of the element, 'value' is the element itself

Types of Iterables:

1. List[]
2. Tuple()
3. String
4. Dictionaries{}
5. Set{}
6. Range()
7. Enumerate()
8. Zip()
9. file

<u>Some example for 'for loop' syntax</u>

Looping through a list:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
```

```
    print(fruit)
```

Looping through a string:

```
for char in "Hello":
    print(char)
```

Looping through a Dictionaries:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}

# Iterating over items in the dictionary
for key, value in my_dict.items():
    print(f"Key: {key}, Value: {value}")

# Performing an operation based on the dictionary values
for key, value in my_dict.items():
    if isinstance(value, str):
        print(f"{key} is a string with length {len(value)}")
    elif isinstance(value, int):
        print(f"{key} is an integer")
    else:
        print(f"{key} has an unknown type")
```

Looping through a file:

```
with open("example.txt", "r") as file:
    for line in file:
        print(f"Line from file: {line.strip()}")
```

Using the range function:

```
for i in range(5):
    print(i)
```

Using the enumerate function:

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

Looping with multiple iterables / Using the zip function:

```
fruits = ["apple", "banana", "cherry"]
colors = ["red", "yellow", "red"]
for fruit, color in zip(fruits, colors):
    print(f"Fruit: {fruit}, Color: {color}")
```

Nested Iterables:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for element in row:
        print(f"Element in matrix: {element}")
```

Example 1

```
print("2 table")
for i in range(1,11):
```

```
    j = i*2
    print(i,"X 2 =",j)
    print("{} X 2 = {}".format(i, j))
```

Example 2

```
print("Even and odd numbers count from 1 to 10")
evencount=0
oddcount=0
for i in range(1,11):
    if(i%2==0):
        evencount=evencount+1
    else:
        oddcount=oddcount+1
print("even :", evencount)
print("odd :", oddcount)
```

Functions

Defining a Function:

```
def function_name(parameters):
    """
    Docstring: Optional documentation for the function
    """
    # Function body: Code that performs the task
    # ...
    return result  # Optional return statement
```

- def: Keyword used to define a function.
- function_name: Name of the function (follows the same naming rules as variables).
- parameters: Input values that the function can accept (optional).
    - Positional Parameters
    - default values
    - Variable Positional Parameters (*args)
    - Variable Keyword Parameters (**kwargs)
- Docstring: Optional documentation string providing information about the function.
- return: (optional) Keyword used to specify the value that the function should return.

Calling a function:

```
result = function_name(arguments)
```

<u>Some example for functions syntax</u>

Example 1

```
def complex_function(arg1, *args, kwarg1="default_value", **kwargs):
```

```
    """
    A function with a mix of positional arguments, variable positional arguments,
    a keyword argument with a default value, and variable keyword arguments.
    """
    print("arg1:", arg1)
    print("args:", args)
    print("kwarg1:", kwarg1)
    print("kwargs:", kwargs)

# Example call
complex_function(1, 2, 3, kwarg1="custom_value", option1="value1", option2="value2")
```

Example 2

```
def place_order(item, quantity=1, *additional_items, shipping_method="Standard",
**special_requests):
    """
    Simulates placing an online order with various parameters.
    """
    print("Placing order:")
    print(f"Item: {item}")
    print(f"Quantity: {quantity}")
    if additional_items:
        print("Additional Items:", additional_items)
    print(f"Shipping Method: {shipping_method}")
    if special_requests:
        print("Special Requests:", special_requests)
    print("Order placed successfully!\n")


# Example Calls:
# Basic order with default values
place_order("Laptop")

# Order with additional items, custom quantity, and expedited shipping
place_order("Smartphone", 2, "Headphones", "Charger", shipping_method="Expedited")

# Order with special requests
place_order("Book", special_instructions="Gift wrap", delivery_notes="Call before delivery")
```

Example 3

```
def add():
    print("Addition")
    a = int(input("Enter a vale:"))
    b = int(input("Enter b vale:"))
    print(a+b)

def sub():
    print("Subtraction")
    a = int(input("Enter a vale:"))
```

```
    b = int(input("Enter b vale:"))
    print(a-b)

print("Enter add or sub")
i=input()
if(i=="add"):
    add()
else:
    sub()
```

Class

1. Regular (Simple) Class:

Defining a Class:

```
class ClassName:
    # Class variables (shared among all instances)
    class_variable = value

    # Constructor method (initializer)
    def __init__(self, parameter1, parameter2, ...):
        # Instance variables (unique to each instance)
        self.instance_variable1 = parameter1
        self.instance_variable2 = parameter2
        # ... additional initialization code

    # Instance methods
    def method1(self, ...):
        # code for method1
        pass

    def method2(self, ...):
        # code for method2
        pass
    # ... additional methods
```

Calling a class:

```
# Creating instances of MyClass
obj1 = ClassName ("argument 1, argument 2,..")
obj2 = ClassName ("argument 101, argument 102,..")

# Calling instance method
obj1. method1 ()
obj1. Method2 ()
obj2. method1 ()
obj2. Method2 ()
```

- **class ClassName:**: This line starts the class definition. Replace "ClassName" with the desired name for your class.

- **class_variable = value**: Class variables are shared among all instances of the class. You can define them directly within the class body.
- **def __init__(self, parameter1, parameter2, ...):**: The __init__ method is a special method called the constructor. It initializes the object's attributes when an instance of the class is created. self is a reference to the instance being created.
- **self.instance_variable1 = parameter1:** These lines inside the __init__ method create instance variables. They are unique to each instance of the class and store specific information about that instance.
- **def method1(self, ...):**: These are instance methods. They define the behaviors of the class. The first parameter, self, is a reference to the instance on which the method is called.
- To use the class:
    - Create instances (obj1 and obj2) by calling the class with parentheses.
    - Access the instance variables and call instance methods using the created objects.

Example 1:

```
class Dog:
    # Class variable
    species = "Canis familiaris"

    # Constructor method (initializer)
    def __init__(self, name, age):
        # Instance variables
        self.name = name
        self.age = age

    # Instance method
    def bark(self):
        print(f"{self.name} says Woof!")

# Creating instances of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Molly", 5)

# Accessing instance variables
print(f"{dog1.name} is {dog1.age} years old.")
print(f"{dog2.name} is {dog2.age} years old.")

# Calling instance method
dog1.bark()
dog2.bark()

# Accessing class variable
print(f"They both belong to the species {Dog.species}.")
```

2. Inheritance (Subclass) Class:

Defining Class:

```
class BaseClass:
    def __init__(self, base_param):
```

```
        self.base_param = base_param

    def base_method(self):
        print("Base method")

# Subclass inheriting from BaseClass
class SubClass(BaseClass):
    def __init__(self, base_param, sub_param):
        # Call the constructor of the base class
        super().__init__(base_param)
        self.sub_param = sub_param

    def sub_method(self):
        print("Subclass method")
```

Calling a class:

```
# Creating instances of the subclasses
base_instance = BaseClass("Base Parameter")
sub_instance = SubClass("Base Parameter", "Sub Parameter")

# Accessing attributes and calling methods
print(base_instance.base_param)
base_instance.base_method()

print(sub_instance.base_param)
print(sub_instance.sub_param)
sub_instance.base_method()
sub_instance.sub_method()
```

- BaseClass is the base class with a constructor (__init__) and a method (base_method).
- SubClass is a subclass of BaseClass and inherits its attributes and methods.
- The super().__init__(base_param) call in SubClass's constructor invokes the constructor of the base class.
- The sub_method is a method specific to the subclass.
- Instances of both classes are created, and you can access attributes and call methods on them.

Example 2:

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

    def speak(self):
        pass  # Abstract method, to be overridden in subclasses

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"
```

```
class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# Creating instances of subclasses
dog_instance = Dog("Buddy")
cat_instance = Cat("Whiskers")

# Calling methods of the subclasses
print(dog_instance.speak())  # Output: Buddy says Woof!
print(cat_instance.speak())  # Output: Whiskers says Meow!
```

Error handling

1. **try** - The try block contains the code that might raise an exception.
2. **except** - The except blocks catch specific exceptions. If an exception is caught, the corresponding block is executed.
3. **else** - The else block is executed if no exception occurs in the try block. It prints the result of the division.
4. **finally** - The finally block is always executed, regardless of whether an exception occurred or not. It is often used for cleanup operations.

Base Exception

1. Exception
    1) SyntaxErrors:
    2) AttributeError
    3) ArithmeticError
        i. ZeroDivisionError
        ii. OverflowError
        iii. FloatingPointError
    4) EOFError
    5) NameError
    6) LookupError
        i. IndexError
        ii. KeyError
    7) OSError
        i. FileNotFoundError
        ii. InterruptedError
        iii. PermissionError
        iv. TimeOutError
    8) TypeError
    9) ValueError
2. KeyboardInterrupt

Note: There are always more specific exceptions and errors that may not be covered in a general overview. Python's standard library and third-party libraries can introduce additional exceptions based on the functionality they provide.

Example 1

```
def divide_numbers(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
    except TypeError as e:
        print(f"Error: {e}")
    else:
        print(f"The result of {x} / {y} is: {result}")
    finally:
        print("This block always executes, regardless of whether an exception occurred or not.")

# Example 1: Valid division
divide_numbers(10, 2)

# Example 2: Division by zero
divide_numbers(5, 0)

# Example 3: Type error
divide_numbers("10", 2)
```

Example 2

```
try:
        number = int(input("Enter a number: "))
        print(f"Your entered number is: {number}")
    except ValueError:
        print("Error: Please enter a valid integer.")
```

---

keywords in python

Keywords in Python are reserved words that have special meanings and cannot be used as identifiers (such as variable names, function names, or class names). These keywords are an integral part of the Python language and are used to define the syntax and structure of the code. Here is a list of Python keywords:

Note: Keywords are case sensitive.

1. True
2. False
3. and
4. not
5. or
6. as

7. in
8. is
9. if
10. elif
11. else
12. for
13. while
14. class
15. def
16. lambda
17. break
18. continue
19. pass
20. return
21. try
22. except
23. finally
24. None
25. import
26. del
27. from
28. with
29. assert
30. async
31. await
32. global
33. nonlocal
34. raise
35. yield

These keywords have specific meanings in the Python language, and they should not be used as names for variables, functions, or classes in your code.

---

escape characters

1. \n: Newline character - used to start a new line.
2. \t: Tab character - used to create horizontal spacing.
3. \' and \": Single and double quote characters, respectively.
4. \\: Backslash character - used to represent a literal backslash.
5. \r: Carriage return character - used to move the cursor to the beginning of the line.
6. \b: Backspace character - used to move the cursor one position back.
7. \f: Form feed character - used for page breaks.
8. \v: Vertical tab character.

Note: The list contains the most frequently used escape characters in Python. Cover most use cases.

File handling

1. Opening a File - **open()**
    - **file = open('example.txt', 'r')** # Opening a file for reading
    - **file = open('example.txt', 'w')** # Opening a file for writing
    - **file = open('example.txt', 'a')** # Opening a file for appending
2. Writing to a File - **write()**
    - **file.write("Hello, this is a line of text.")**
3. Reading from a File –
    - **read():** Reads the entire content of the file.
        - **content = file.read()**
    - **readline():** Reads one line from the file.
        - **line = file.readline()**
    - **readlines():** Reads all lines from the file and returns them as a list.
        - **lines = file.readlines()**
4. Closing a File - **close()**
    - **file.close()**
5. Using '**with**' Statement (Context Manager) - File is automatically closed when exiting the 'with' block
    - **with open('example.txt', 'r') as file**:
      content = file.read()
6. Appending to a File - To append content to an existing file, open the file in append mode (**'a'**).
    - **with open('example.txt', 'a') as file:**
      file.write("This line will be appended.")

Built-in functions

1. Type Conversion:
    - int()
    - float()
    - str()
    - bool()
2. Math Functions:
    - abs()
    - round()
    - max()
    - min()
    - sum()
3. Sequence-related Functions:
    - len()
    - sorted()
    - reversed()
4. Container Functions:

- list()
- tuple()
- set()
- dict()

5. Input/Output Functions:
    - print()
    - input()
6. Logical and Comparison Functions:
    - all()
    - any()
    - bool()
7. Iterating Functions:
    - range()
    - enumerate()
    - zip()
8. File Handling Functions:
    - open()
    - read()
    - write()
    - close()
9. Other Utility Functions:
    - type()
    - id()
    - help()
    - dir()
10. Type-checking functions
    - isinstance()

filter()
map()

Note: These are just a few examples of the many built-in functions available in Python.

Print

1. Printing a string
    - **print("Hello, World!")**
2. Printing with escape characters
    - **print("This is a new line.\nThis is another line.")**
3. Printing multiple values
    - **print("Name:", name, "Age:", age)**
        - name = "John"
        - age = 25
4. Printing with formatting
    - **print("My name is {} and I am {} years old.".format(name, age))**
    - **print("My name is {name} and I am {age} years old.".format)**
5. Printing with f-strings

- **print(f"My name is {name} and I am {age} years old.")**

6. Separator (sep)
   - **print("apple", "orange", "banana", sep=", ")**
7. End (end)
   - **print("Hello", end=" ")**
8. Printing with * operator - to unpack elements from a collection (e.g., a list or tuple) and print them as separate arguments
   - **print(*fruits)**
     - fruits = ["apple", "orange", "banana"]
9. File (file)
   - with open("output.txt", "w") as f:
     print("This is written to a file.", file=f)

Example 1: Printing Multiple Values with Formatting and Separator

```
name = "John"
age = 25
city = "New York"

print("Name:", name, "Age:", age, "City:", city, sep=" | ")
```

Example 2: Printing with F-strings and End

```
item = "Python Book"
price = 29.99

print(f"Item: {item}, Price: ${price:.2f}", end=" - Available Now!\n")
```

Input()

1. Prompting with a Message
   - **user_input = input("Enter something: ")**
2. Converting Input to a Specific Type
   - **num = int(input("Enter an integer: "))**
3. Handling Empty Input or Defaults
   - **user_input = input("Enter something (or press Enter for default): ") or "default_value"**
4. Using Strip to Remove Whitespace
   - **user_input = input("Enter something: ").strip()**
5. Multiple Inputs
   - **values = input("Enter two values separated by a space: ").split()**
   - value1, value2 = values[0], values[1]

Set = {element1, element2, element3,..}

```
a = {1, 2, 3, 4, 5, 5}      #no dublicate vales
print(a)
#a[1] = 44                  #can't replace
a.add(10)
print(a)
a.remove(4)
print(a)
a.pop()                     #set is unorder
print(a)
b = {56, 67, 34}
a.update(b)
print(a)
```