**CONDITION AND BRANCHING**

If (condition):

Execution of code if the condition is true

elif (condition):

Execution of code if the condition is true

Else:

Execution of code if the condition is false

**LOOP**

colors = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]

for color in colors:

print(color)

for number in range(1, 11):

print(number)

**Range Function**

The range function in Python generates an ordered sequence that can be used in loops. It takes one or two arguments:

* If given one argument (e.g., range(11)), it generates a sequence starting from 0 up to (but not including) the given number.

1. for number in range(11):
2. print(number)

* If given two arguments (e.g., range(1, 11)), it generates a sequence starting from the first argument up to (but not including) the second argument.

1. for number in range(1, 11):
2. print(number)

**The Enumerated For Loop**

Have you ever needed to keep track of both the item and its position in a list? An enumerated for loop comes to your rescue. It's like having a personal assistant who not only hands you the item but also tells you where to find it.

Consider this example:

1. fruits = ["apple", "banana", "orange"]
2. for index, fruit in enumerate(fruits):
3. print(f"At position {index}, I found a {fruit}")

With this loop, you not only get the fruit but also its position in the list. It's as if you have a magical guide pointing out each fruit's location!

**While Loops**

While loops are like a sleepless night at a friend's sleepover. Imagine you and your friends keep telling ghost stories until someone decides it's time to sleep. As long as no one says, "Let's sleep" you keep telling stories.  
A while loop works similarly - it repeats a task as long as a certain condition is true. It's like saying, "Hey computer, keep doing this until I say stop!"

**Basic syntax of While Loop.**

1. while condition:
2. # Code to be executed while the condition is true
3. # Indentation is crucial to indicate the scope of the loop

For example, here's how you might use a while loop to count from 1 to 10:

1. count = 1
2. while count <= 10:
3. print(count)
4. count += 1

here's a breakdown of the above code.

1. There is a variable named **count** initialized with the value 1.
2. The while loop is used to repeatedly execute a block of code as long as a given condition is True. In this case, the condition is **count <= 10**, meaning the loop will continue as long as count is less than or equal to 10.
3. Inside the loop:
   * The **print(count)** statement outputs the current value of the count variable.
   * The **count += 1** statement increments the value of count by 1. This step ensures that the loop will eventually terminate when count becomes greater than 10.
4. The loop will continue executing as long as the condition count <= 10 is satisfied.
5. The loop will print the numbers 1 to 10 in consecutive order since the print statement is inside the loop block and executed during each iteration.
6. Once count reaches 11, the condition count <= 10 will evaluate to False, and the loop will terminate.
7. The output of the code will be the numbers 1 to 10, each printed on a separate line.

**The Loop Flow**

Both for and while loops have their special moves, but they follow a pattern:

* **Initialization:** You set up things like a starting point or conditions.
* **Condition:** You decide when the loop should keep going and when it should stop.
* **Execution:** You do the task inside the loop.
* **Update:** You make changes to your starting point or conditions to move forward.
* **Repeat:** The loop goes back to step 2 until the condition is no longer true.

**When to Use Each**

**For Loops:** Use for loops when you know the number of iterations in advance and want to process each element in a sequence. They are best suited for iterating over collections and sequences where the length is known.

**While Loops:** Use while loops when you need to perform a task repeatedly as long as a certain condition holds true. While loops are particularly useful for situations where the number of iterations is uncertain or where you're waiting for a specific condition to be met.

**FUNCTIONS**

Def function(a):

“add 1 to a””

B =a+1

Print (a, “+1 = ”, b)

Return b

**Purpose of functions**

Functions promote code modularity and reusability. Imagine you have a task that needs to be performed multiple times within a program. Instead of duplicating the same code at various places, you can define a function once and call it whenever you need that task. This reduces redundancy and makes the code easier to manage and maintain.

**Benefits of using functions**

**Modularity:** Functions break down complex tasks into manageable components  
**Reusability:** Functions can be used multiple times without rewriting code  
**Readability:** Functions with meaningful names enhance code understanding  
**Debugging:**Isolating functions eases troubleshooting and issue fixing  
**Abstraction:** Functions simplify complex processes behind a user-friendly interface  
**Collaboration:** Team members can work on different functions concurrently  
**Maintenance:** Changes made in a function automatically apply wherever it's used

**How functions take inputs, perform tasks, and produce outputs**

**Inputs (Parameters)**

Functions operate on data, and they can receive data as input. These inputs are known as *parameters* or *arguments*. Parameters provide functions with the necessary information they need to perform their tasks. Consider parameters as values you pass to a function, allowing it to work with specific data.

**Performing tasks**

Once a function receives its input (parameters), it executes predefined actions or computations. These actions can include calculations, operations on data, or even more complex tasks. The purpose of a function determines the tasks it performs. For instance, a function could calculate the sum of numbers, sort a list, format text, or fetch data from a database.

**Producing outputs**

After performing its tasks, a function can produce an output. This output is the result of the operations carried out within the function. It's the value that the function “returns” to the code that called it. Think of the output as the end product of the function's work. You can use this output in your code, assign it to variables, pass it to other functions, or even print it out for display.

Example:

Consider a function named calculate\_total that takes two numbers as input (parameters), adds them together, and then produces the sum as the output. Here's how it works:

1. def calculate\_total(a, b): # Parameters: a and b
2. total = a + b # Task: Addition
3. return total # Output: Sum of a and b
4. result = calculate\_total(5, 7) # Calling the function with inputs 5 and 7
5. print(result) # Output: 12

**Python's built-in functions**

Python has a rich set of built-in functions that provide a wide range of functionalities. These functions are readily available for you to use, and you don't need to be concerned about how they are implemented internally. Instead, you can focus on understanding what each function does and how to use it effectively.

**Using built-in functions or Pre-defined functions**

To use a built-in function, you simply call the function's name followed by parentheses. Any required arguments or parameters are passed into the function within these parentheses. The function then performs its predefined task and may return an output you can use in your code.

Here are a few examples of commonly used built-in functions:

**len():** Calculates the length of a sequence or collection

1. string\_length = len("Hello, World!") # Output: 13
2. list\_length = len([1, 2, 3, 4, 5]) # Output: 5

**sum():** Adds up the elements in an iterable (list, tuple, and so on)

1. total = sum([10, 20, 30, 40, 50]) # Output: 150

**max():** Returns the maximum value in an iterable

1. highest = max([5, 12, 8, 23, 16]) # Output: 23

**min():** Returns the minimum value in an iterable

1. lowest = min([5, 12, 8, 23, 16]) # Output: 5

Python's built-in functions offer a wide array of functionalities, from basic operations like len() and sum() to more specialized tasks.

**Defining your functions**

Defining a function is like creating your mini-program:

1. Use def followed by the function name and parentheses

Here is the syntax to define a function:

1. def function\_name():
2. pass

A "pass" statement in a programming function is a placeholder or a no-op (no operation) statement. Use it when you want to define a function or a code block syntactically but do not want to specify any functionality or implementation at that moment.

* **Placeholder:** "pass" acts as a temporary placeholder for future code that you intend to write within a function or a code block.
* **Syntax Requirement:** In many programming languages like Python, using "pass" is necessary when you define a function or a conditional block. It ensures that the code remains syntactically correct, even if it doesn't do anything yet.
* **No Operation:** "pass" itself doesn't perform any meaningful action. When the interpreter encounters “pass”, it simply moves on to the next statement without executing any code.

**Function Parameters:**

* Parameters are like inputs for functions
* They go inside parentheses when defining the function
* Functions can have multiple parameters

Example:

1. def greet(name):
2. print("Hello, " + name)
3. result = greet("Alice")
4. print(result) # Output: Hello, Alice

**Docstrings (Documentation Strings)**

* Docstrings explain what a function does
* Placed inside triple quotes under the function definition
* Helps other developers understand your function

Example:

1. def multiply(a, b):
2. """
3. This function multiplies two numbers.
4. Input: a (number), b (number)
5. Output: Product of a and b
6. """
7. print(a \* b)
8. multiply(2,6)

**Return statement**

* Return gives back a value from a function
* Ends the function's execution and sends the result
* A function can return various types of data

Example:

1. def add(a, b):
2. return a + b
3. sum\_result = add(3, 5) # sum\_result gets the value 8

**Understanding scopes and variables**

Scope is where a variable can be seen and used:

* **Global Scope:** Variables defined outside functions; accessible everywhere
* **Local Scope:** Variables inside functions; only usable within that function

Example:

**Part 1: Global variable declaration**

1. global\_variable = "I'm global"

This line initializes a global variable called global\_variable and assigns it the value "I'm global".

*Global variables are accessible throughout the entire program, both inside and outside functions.*

**Part 2: Function definition**

1. def example\_function():
2. local\_variable = "I'm local"
3. print(global\_variable) # Accessing global variable
4. print(local\_variable) # Accessing local variable

Here, you define a function called example\_function().

Within this function:

* A local variable named local\_variable is declared and initialized with the string value "I'm local." This variable is local to the function and can only be accessed within the function's scope.
* The function then prints the values of both the **global variable (global\_variable) and the local variable (local\_variable)**. It demonstrates that you can access global and local variables within a function.

**Part 3: Function call**

1. example\_function()

In this part, you call the example\_function() by invoking it. This results in the function's code being executed.  
As a result of this function call, it will print the values of the global and local variables within the function.

**Part 4: Accessing global variable outside the function**

1. print(global\_variable) # Accessible outside the function

After calling the function, you print the value of the global variable global\_variable outside the function. **This demonstrates that global variables are accessible inside and outside of functions.**

**Part 5: Attempting to access local variable outside the function**

1. # print(local\_variable) # Error, local variable not visible here

In this part, you are attempting to print the value of the local variable local\_variable outside of the function. However, this line would result in an error.

*Local variables are only visible and accessible within the scope of the function where they are defined.*

Attempting to access them outside of that scope would raise a "NameError".

**Using functions with loops**

**Functions and loops together**

1. Functions can contain code with loops
2. This makes complex tasks more organized
3. The loop code becomes a repeatable function

Example:

1. def print\_numbers(limit):
2. for i in range(1, limit+1):
3. print(i)
4. print\_numbers(5) # Output: 1 2 3 4 5

**Enhancing code organization and reusability**

1. Functions group similar actions for easy understanding
2. Looping within functions keeps code clean
3. You can reuse a function to repeat actions

Example

1. def greet(name):
2. return "Hello, " + name
3. for \_ in range(3):
4. print(greet("Alice"))

**Modifying data structure using functions**

You'll use Python and a list as the data structure for this illustration. In this example, you will create functions to add and remove elements from a list.

**Part 1: Initialize an empty list**

1. # Define an empty list as the initial data structure
2. my\_list = []

In this part, you start by creating an empty list named my\_list. This empty list serves as the data structure that you will modify throughout the code.

**Part 2: Define a function to add elements**

1. # Function to add an element to the list
2. def add\_element(data\_structure, element):
3. data\_structure.append(element)

Here, you define a function called add\_element. This function takes two parameters:

* data\_structure: This parameter represents the list to which you want to add an element
* element: This parameter represents the element you want to add to the list

Inside the function, you use the append method to add the provided element to the data\_structure, which is assumed to be a list.

**Part 3: Define a function to remove elements**

1. # Function to remove an element from the list
2. def remove\_element(data\_structure, element):
3. if element in data\_structure:
4. data\_structure.remove(element)
5. else:
6. print(f"{element} not found in the list.")

In this part, you define another function called remove\_element. It also takes two parameters:

* data\_structure: The list from which we want to remove an element
* element: The element we want to remove from the list

Inside the function, you use conditional statements to check if the element is present in the data\_structure. If it is, you use the remove method to remove the first occurrence of the element. If it's not found, you print a message indicating that the element was not found in the list.

**Part 4: Add elements to the list**

1. # Add elements to the list using the add\_element function
2. add\_element(my\_list, 42)
3. add\_element(my\_list, 17)
4. add\_element(my\_list, 99)

Here, you use the add\_element function to add three elements (42, 17, and 99) to the my\_list. These are added one at a time using function calls.

**Part 5: Print the current list**

1. # Print the current list
2. print("Current list:", my\_list)

This part simply prints the current state of the my\_list to the console, allowing us to see the elements that have been added so far.

**Part 6: Remove elements from the list**

1. # Remove an element from the list using the remove\_element function
2. remove\_element(my\_list, 17)
3. remove\_element(my\_list, 55) # This will print a message since 55 is not in the list

In this part, you use the remove\_element function to remove elements from the my\_list. First, you attempt to remove 17 (which is in the list), and then you try to remove 55 (which is not in the list). **The second call to remove\_element will print a message indicating that 55 was not found.**

**Part 7: Print the updated list**

1. # Print the updated list
2. print("Updated list:", my\_list)

Finally, you print the updated my\_list to the console. This allows us to observe the modifications made to the list by adding and removing elements using the defined functions.

**EXCEPTION HANDLING**

**Errors vs. Exceptions**

Hold on, what is the difference between errors and exceptions? Well, errors are usually big problems that come from the computer or the system. They often make the program stop working completely. On the other hand, exceptions are more like issues we can control. They happen because of something we did in our code and can usually be fixed, so the program keeps going.

Here is the difference between Errors and exceptions:-

| **Aspect** | **Errors** | **Exceptions** |
| --- | --- | --- |
| **Origin** | Errors are typically caused by the | Exceptions are usually a result of |
|  | environment, hardware, or operating system. | problematic code execution within the program. |
| **Nature** | Errors are often severe and can lead to | Exceptions are generally less severe and can |
|  | program crashes or abnormal termination. | be caught and handled to prevent program |
|  |  | termination. |
| **Handling** | Errors are not usually caught or handled | Exceptions can be caught using try-except |
|  | by the program itself. | blocks and dealt with gracefully, allowing |
|  |  | the program to continue execution. |
| **Examples** | Examples include “SyntaxError” due to | Examples include “ZeroDivisionError” when |
|  | incorrect syntax or “NameError” when a | dividing by zero, or “FileNotFoundError” when |
|  | variable is not defined. | attempting to open a non-existent file. |
| **Categorization** | Errors are not classified into categories. | Exceptions are categorized into various |
|  |  | classes, such as “ArithmeticError,” “IOError,” |
|  |  | “ValueError,” etc., based on their nature. |

**Common Exceptions in Python**

Here are a few examples of exceptions we often run into and can handle using this tool:

* **ZeroDivisionError:** This error arises when an attempt is made to divide a number by zero. Division by zero is undefined in mathematics, causing an arithmetic error. For instance:  
  For example:

1. result = 10 / 0
2. print(result)
3. # Raises ZeroDivisionError

* **ValueError:** This error occurs when an inappropriate value is used within the code. An example of this is when trying to convert a non-numeric string to an integer:  
  For example:

1. num = int("abc")
2. # Raises ValueError

* **FileNotFoundError:** This exception is encountered when an attempt is made to access a file that does not exist.  
  For example:

1. with open("nonexistent\_file.txt", "r") as file:
2. content = file.read() # Raises FileNotFoundError

* **IndexError:** An IndexError occurs when an index is used to access an element in a list that is outside the valid index range.  
  For example:

1. my\_list = [1, 2, 3]
2. value = my\_list[1] # No IndexError, within range
3. missing = my\_list[5] # Raises IndexError

* **KeyError:** The KeyError arises when an attempt is made to access a non-existent key in a dictionary.  
  For example:

1. my\_dict = {"name": "Alice", "age": 30}
2. value = my\_dict.get("city") # No KeyError, using .get() method
3. missing = my\_dict["city"] # Raises KeyError

* **TypeError:** The TypeError occurs when an object is used in an incompatible manner. An example includes trying to concatenate a string and an integer:  
  For example:

1. result = "hello" + 5
2. # Raises TypeError

* **AttributeError:** An AttributeError occurs when an attribute or method is accessed on an object that doesn't possess that specific attribute or method. For instance:  
  For example:

1. text = "example"
2. length = len(text) # No AttributeError, correct method usage
3. missing = text.some\_method() # Raises AttributeError

* **ImportError:** This error is encountered when an attempt is made to import a module that is unavailable. For example: import non\_existent\_module

*Note:****Please remember, the exceptions you will encounter are not limited to just these. There are many more in Python. However, there is no need to worry. By using the technique provided below and following the correct syntax, you will be able to handle any exceptions that come your way.***

**Handling Exceptions:**

Python has a handy tool called try and except that helps us manage exceptions.

**Try and Except** : You can use the try and except blocks to prevent your program from crashing due to exceptions.

Here's how they work:

1. The code that may result in an exception is contained in the try block.
2. If an exception occurs, the code directly jumps to except block.
3. In the except block, you can define how to handle the exception gracefully, like displaying an error message or taking alternative actions.
4. After the except block, the program continues executing the remaining code.

**Example: Attempting to divide by zero**

1. # using Try- except
2. try:
3. # Attempting to divide 10 by 0
4. result = 10 / 0
5. except ZeroDivisionError:
6. # Handling the ZeroDivisionError and printing an error message
7. print("Error: Cannot divide by zero")
8. # This line will be executed regardless of whether an exception occurred
9. print("outside of try and except block")

**OBJECTS AND CLASSES**

**Introduction to Classes and object**

Python is an object-oriented programming (OOP) language, which means it uses a paradigm centered around objects and classes. Here's an explanation of these fundamental concepts:

**Classes:**

A class is a blueprint or template for creating objects. It defines the structure and behavior that its objects will have.

Think of a class as a cookie cutter, and objects as the cookies cut from that template.

In Python, classes are created using the class keyword.

**Creating Classes:**

When you create a class, you specify the attributes(data) and methods (functions) that objects of that class will have.  
Attributes are defined as variables within the class, and methods are defined as functions.  
For example,you can design a "Car" class with attributes such as "color" and "speed," along with methods like "accelerate."

**Objects:**

An *object* is a fundamental unit in Python that represents a real-world entity or concept.  
Objects can be tangible (like a car) or abstract (like a student's grade).

*Every object has two main characteristics:*

**State:**

The *attributes or data* that describe the object. For our "Car" object, this might include attributes like "color", "speed", and "fuel level".

**Behavior:**

The *actions or methods* that the object can perform. In Python, methods are functions that belong to objects and can change the object's state or perform specific operations.

**Instantiating Objects:**

* Once you've defined a class, you can create individual objects (instances) based on that class.
* Each object is independent and has its own set of attributes and methods.
* To create an object, you use the class name followed by parentheses, so: "my\_car = Car()"

**Interacting with Objects:**

You interact with objects by calling their methods or accessing their attributes using dot notation.

For example, if you have a Car object named **my\_car**, you can set its color with **my\_car.color = "blue"** and accelerate it with **my\_car.accelerate()** if there's an accelerate method defined in the class.

**Structure of classes and object code**

*Please don't directly copy and use this code as it's meant as a template for explanation and isn't tailored for specific results.*

**Class Declaration (class ClassName):**

* The class keyword is used to declare a class in Python.
* ClassName is the name of the class, typically following CamelCase naming conventions.

1. class ClassName:

**Class Attributes (class\_attribute = value):**

* Class attributes are variables that are shared among all instances (objects) of the class.
* They are defined within the class but outside of any methods.

1. class ClassName:
2. # Class attributes (shared by all instances)
3. class\_attribute = value

**Constructor Method (def init(self, attribute1, attribute2, …):):**

* The \_\_init\_\_ method is a special method known as the constructor.
* It initializes the **instance attributes** (also called instance variables) when an object is created.
* The self parameter is the first parameter of the constructor, referring to the instance being created.
* **attribute1**, **attribute2**, and so on are parameters passed to the constructor when creating an object.
* Inside the constructor, self.attribute1, self.attribute2, and so on are used to assign values to instance attributes.

1. class ClassName:
2. # Class attributes (shared by all instances)
3. class\_attribute = value
4. # Constructor method (initialize instance attributes)
5. def \_\_init\_\_(self, attribute1, attribute2, ...):
6. pass
7. # ...

**Instance Attributes (self.attribute1 = attribute1):**

* Instance attributes are variables that store data specific to each instance of the class.
* They are initialized within the **init** method using the self keyword followed by the attribute name.
* These attributes hold unique data for each object created from the class.

1. class ClassName:
2. # Class attributes (shared by all instances)
3. class\_attribute = value
4. # Constructor method (initialize instance attributes)
5. def \_\_init\_\_(self, attribute1, attribute2, ...):
6. self.attribute1 = attribute1
7. self.attribute2 = attribute2
8. # ...

**Instance Methods (def method1(self, parameter1, parameter2, …):):**

* Instance methods are functions defined within the class.
* They operate on the instance's data (instance attributes) and can perform actions specific to instances.
* The **self** parameter is required in instance methods, allowing them to access instance attributes and call other methods within the class.

1. class ClassName:
2. # Class attributes (shared by all instances)
3. class\_attribute = value
4. # Constructor method (initialize instance attributes)
5. def \_\_init\_\_(self, attribute1, attribute2, ...):
6. self.attribute1 = attribute1
7. self.attribute2 = attribute2
8. # ...
9. # Instance methods (functions)
10. def method1(self, parameter1, parameter2, ...):
11. # Method logic
12. pass

*Using same steps you can define multiple instance methods.*

1. class ClassName:
2. # Class attributes (shared by all instances)
3. class\_attribute = value
4. # Constructor method (initialize instance attributes)
5. def \_\_init\_\_(self, attribute1, attribute2, ...):
6. self.attribute1 = attribute1
7. self.attribute2 = attribute2
8. # ...
9. # Instance methods (functions)
10. def method1(self, parameter1, parameter2, ...):
11. # Method logic
12. pass
13. def method2(self, parameter1, parameter2, ...):
14. # Method logic
15. pass

*Note:- Now, we have successfully created a dummy class.*

**Creating Objects (Instances):**

* To create objects (instances) of the class, you call the class like a function and provide arguments required by the constructor.
* Each object is a distinct instance of the class, with its own set of instance attributes and the ability to call methods defined in the class.

1. 1
2. 2
3. 3
4. # Create objects (instances) of the class
5. object1 = ClassName(arg1, arg2, ...)
6. object2 = ClassName(arg1, arg2, ...)

**Calling methods on objects:**

* In this section we will call methods on objects, specifically object1 and object2.
* The methods **method1** and **method2** are defined in the ClassName **class**, and you're calling them on **object1** and **object2** respectively.
* You pass values **param1\_value** and **param2\_value** as arguments to these methods. These arguments are used within the method's logic.

**Method 1: Using dot notation:**

* This is the most straightforward way to call an object's method. In this we use the dot notation **(object.method())** to directly invoke the method on the object.
* For example, result1 = object1.method1(param1\_value, param2\_value, ...) calls method1 on object1.

1. # Calling methods on objects
2. # Method 1: Using dot notation
3. result1 = object1.method1(param1\_value, param2\_value, ...)
4. result2 = object2.method2(param1\_value, param2\_value, ...)

**Method 2: Assigning object methods to variables:**

* Here's an alternative way to call an object's method by assigning the method reference to a variable.
* method\_reference = object1.method1 assigns the method **method1** of **object1** to the variable **method\_reference**.
* Later, we call the method using the variable like this: **result3 = method\_reference(param1\_value, param2\_value, …)**.

1. # Method 2: Assigning object methods to variables
2. method\_reference = object1.method1 # Assign the method to a variable
3. result3 = method\_reference(param1\_value, param2\_value, ...)

**Accessing object attributes:**

* Here, we are accessing an object's attribute using dot notation.
* attribute\_value = object1.attribute1 retrieves the value of the attribute **attribute1** from **object1** and assigns it to the variable **attribute\_value**.

1. # Accessing object attributes
2. attribute\_value = object1.attribute1 # Access the attribute using dot notation

**Modifying object attributes:**

* We are modifying an object's attribute using dot notation.
* object1.attribute2 = new\_value sets the attribute **attribute2** of **object1** to the new value **new\_value**.

1. # Modifying object attributes
2. object1.attribute2 = new\_value # Change the value of an attribute using dot notation

**Accessing class attributes (shared by all instances):**

* Finally,we access a class attribute, which is shared by all instances of the class.
* class\_attr\_value = ClassName.class\_attribute accesses the class attribute class\_attribute from the ClassName class and assigns its value to the variable  
  class\_attr\_value.

1. # Accessing class attributes (shared by all instances)
2. class\_attr\_value = ClassName.class\_attribute

**Real-world example**

Let's write a python program that simulates a simple car class, allowing you to create car instances, accelerate them, and display their current speeds.

1. Let's start by defining a Car class that includes the following attributes and methods:

* Class attribute max\_speed, which is set to **120 km/h**.
* Constructor method \_\_init\_\_ that takes parameters for the **car's make, model, color, and an optional speed (defaulting to 0)**. This method initializes instance attributes for make, model, color, and speed.
* Method accelerate(self, acceleration) that allows the car to accelerate. If the acceleration does not exceed the max\_speed, update the **car's speed** attribute. Otherwise, set the speed to the **max\_speed**.
* Method get\_speed(self) that returns the current speed of the car.

1. class Car:
2. # Class attribute (shared by all instances)
3. max\_speed = 120 # Maximum speed in km/h
4. # Constructor method (initialize instance attributes)
5. def \_\_init\_\_(self, make, model, color, speed=0):
6. self.make = make
7. self.model = model
8. self.color = color
9. self.speed = speed # Initial speed is set to 0
10. # Method for accelerating the car
11. def accelerate(self, acceleration):
12. if self.speed + acceleration <= Car.max\_speed:
13. self.speed += acceleration
14. else:
15. self.speed = Car.max\_speed
16. # Method to get the current speed of the car
17. def get\_speed(self):
18. return self.speed
19. Now, we will instantiate two objects of the Car class, each with the following characteristics:

* car1: **Make = "Toyota", Model = "Camry", Color = "Blue"**
* car2: **Make = "Honda", Model = "Civic", Color = "Red"**

1. # Create objects (instances) of the Car class
2. car1 = Car("Toyota", "Camry", "Blue")
3. car2 = Car("Honda", "Civic", "Red")
4. Using the accelerate method, we will increase the speed of car1 by 30 km/h and car2 by 20 km/h.
5. # Accelerate the cars
6. car1.accelerate(30)
7. car2.accelerate(20)
8. Lastly, we will display the current speed of each car by utilizing the `get\_speed method.
9. # Print the current speeds of the cars
10. print(f"{car1.make} {car1.model} is currently at {car1.get\_speed()} km/h.")
11. print(f"{car2.make} {car2.model} is currently at {car2.get\_speed()} km/h.")