**Introduction to classes and object**

Python is an object-oriented programming (OOP) language that uses a paradigm centered around objects and classes.

Let's look at 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, you can create classes 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 your "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 because it is a template for explanation and not 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:

Copied!

**Class attributes (class\_attribute = value)**

* Class attributes are variables shared among all class instances (objects).
* 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

Copied!

**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. # ...

Copied!

**Instance attributes (self.attribute1 = attribute1)**

* Instance attributes are variables that store data specific to each class instance.
* 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. # ...

Copied!

**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

Copied!

*Using the 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

Copied!

*Note: Now, you 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 the constructor requires.
* Each object is a distinct instance of the class, with its own instance attributes and the ability to call methods defined in the class.

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

Copied!

**Calling methods on objects**

* In this section, you 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, use the dot notation **(object.method())** to invoke the method on the object directly.
* 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, ...)

Copied!

**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, 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, ...)

Copied!

**Accessing object attributes**

* Here, you 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**

* You will modify 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, access a class attribute shared by all class instances.
* 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

Copied!

**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

Copied!

1. Now, you 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")

Copied!

1. Using the accelerate method, you will increase the speed of car1 by 30 km/h and car2 by 20 km/h.
2. # Accelerate the cars
3. car1.accelerate(30)
4. car2.accelerate(20)

Copied!

1. Lastly, you will display the current speed of each car by utilizing the get\_speed method.
2. # Print the current speeds of the cars
3. print(f"{car1.make} {car1.model} is currently at {car1.get\_speed()} km/h.")
4. print(f"{car2.make} {car2.model} is currently at {car2.get\_speed()} km/h.")

* Python conditions use “if” statements to execute code based on true/false conditions created by comparisons and Boolean expressions.
* Comparison operations require using comparison operators equal to "=", greater than ">", less than "<".
* An exclamation mark "!" is used to define inequalities of a variable.
* You can compare integers, strings, and floats.
* Python branching directs program flow by using conditional statements (for example, if, else, elif) to execute different code blocks based on conditions or tests.
* You can use the "if" statement with conditions to define actions if true.
* To perform actions based on true or false output, you can use the "else" statement with conditions.
* The elif statement allows for additional checks only if the initial condition is false.
* To execute various operations on Boolean values, we use Boolean logic operators.
* Python loops are control structures that automate repetitive tasks and iterate over data structures like lists or dictionaries.
* The range() function generates a sequence of numbers with a specified start, stop, and step value for loops in Python.
* A for loop in Python iterates over a sequence, such as a list, tuple, or string, and executes a block of code for each item in the sequence.
* A while loop in Python executes a block of code as long as a specified condition remains true.
* Python functions are reusable code blocks that perform specific tasks, take input parameters, and often return results, enhancing code modularity and reusability.
* You may or may not have written the codes that are often included in functions.
* Python has a set of built-in functions such as "len" to find the length of a sequence or "sum" to find the total sum of a sequence.
* The "sorted" function creates a new sorted list, while "sort" sorts items in the original list.
* You can also create your own functions in Python.
* To ensure clarity and organization and facilitate understanding and maintenance of the code, developers must document functions using a documentation string enclosed in three quotes.
* The help command will return the documentation defined for a particular function.
* A function can have multiple parameters.
* “No return” statement in the function means that the function will return nothing.
* The "No work" function does not execute any task. You can use the "pass" keyword to meet the requirement of a non-empty body.
* A function will usually perform more than one task.
* In Python, the scope of a variable determines where you can access or modify that variable. Global scope allows access from anywhere, while local scope restricts it to a block or function.
* In Python, a programmer defines a local variable within a specific block or function, which can only be accessed or modified within that block or function.
* In Python, a global variable is a variable defined at the top level of a program that any part of the code can access or modify.
* Exception handling in Python is a mechanism for managing and responding to errors and exceptions that may occur during program execution, preventing them from crashing the program.
* In Python, you use the "try-except" statement to attempt a block of code and specify alternative actions to execute if an error occurs, allowing you to handle exceptions.
* In Python, you use the "try-except-else" statement to attempt a block of code, handle exceptions in the "except" block, and execute code in the "else" block when no exceptions occur.
* Python developers use the "try-except-else-finally" statement to attempt a block of code, catch exceptions in the "except" block, execute code in the "else" block when no exceptions occur, and ensure that the "finally" block always runs, regardless of whether an exception raised or not.
* In Python, objects are instances of classes that encapsulate data and behavior, serving as the foundation for creating and working with various data types and custom data structures.
* To determine the type of an object in Python, you can use the `type()` command.
* Any changes made within the method of the object may result in a change in object type.
* Classes in Python are blueprints for creating objects, defining their attributes and methods, enabling code organization, and object-oriented programming.
* Function "init" is a special method used to initialize data attributes.
* We can create instances of a class in Python.
* Data attributes consist of the data defining the objects.
* Methods are functions that interact and change the data attributes.
* The method has a function that requires the self as well as other parameters.

**Python Programming Fundamentals Cheat Sheet**

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Syntax and Code Example** |
| AND | Returns `True` if both statement1 and statement2 are `True`. Otherwise, returns `False`. | Syntax:   1. statement1 and statement2   Example:   1. marks = 90 2. attendance\_percentage = 87 3. if marks >= 80 and attendance\_percentage >= 85: 4. print("qualify for honors") 5. else: 6. print("Not qualified for honors") 7. # Output = qualify for honors |
| Class Definition | Defines a blueprint for creating objects and defining their attributes and behaviors. | Syntax:   1. class ClassName: # Class attributes and methods   Example:   1. class Person: 2. def \_\_init\_\_(self, name, age): 3. self.name = name 4. self.age = age |
| Define Function | A `function` is a reusable block of code that performs a specific task or set of tasks when called. | Syntax:   1. def function\_name(parameters): # Function body   Example:   1. def greet(name): print("Hello,", name) |
| Equal(==) | Checks if two values are equal. | Syntax:   1. variable1 == variable2   Example 1:   1. 5 == 5   returns True  Example 2:   1. age = 25 age == 30   returns False |
| For Loop | A `for` loop repeatedly executes a block of code for a specified number of iterations or over a sequence of elements (list, range, string, etc.). | Syntax:   1. for variable in sequence: # Code to repeat   Example 1:   1. for num in range(1, 10): 2. print(num)   Example 2:   1. fruits = ["apple", "banana", "orange", "grape", "kiwi"] 2. for fruit in fruits: 3. print(fruit) |
| Function Call | A function call is the act of executing the code within the function using the provided arguments. | Syntax:   1. function\_name(arguments)   Copied!  Example:   1. greet("Alice") |
| Greater Than or Equal To(>=) | Checks if the value of variable1 is greater than or equal to variable2. | Syntax:   1. variable1 >= variable2   Example 1:   1. 5 >= 5 and 9 >= 5   Copied!  returns True  Example 2:   1. quantity = 105 2. minimum = 100 3. quantity >= minimum   returns True |
| Greater Than(>) | Checks if the value of variable1 is greater than variable2. | Syntax:   1. variable1 > variable2   Example 1: 9 > 6  returns True  Example 2:   1. age = 20 2. max\_age = 25 3. age > max\_age   returns False |
| If Statement | Executes code block `if` the condition is `True`. | Syntax:   1. if condition: #code block for if statement   Example:   1. if temperature > 30: 2. print("It's a hot day!") |
| If-Elif-Else | Executes the first code block if condition1 is `True`, otherwise checks condition2, and so on. If no condition is `True`, the else block is executed. | Syntax:   1. if condition1: 2. # Code if condition1 is True 3. elif condition2: 4. # Code if condition2 is True 5. else: 6. # Code if no condition is True   Example:   1. score = 85 # Example score 2. if score >= 90: 3. print("You got an A!") 4. elif score >= 80: 5. print("You got a B.") 6. else: 7. print("You need to work harder.") 8. # Output = You got a B. |
| If-Else Statement | Executes the first code block if the condition is `True`, otherwise the second block. | Syntax:   1. if condition: # Code, if condition is True 2. else: # Code, if condition is False   Example:   1. if age >= 18: 2. print("You're an adult.") 3. else: 4. print("You're not an adult yet.") |
| Less Than or Equal To(<=) | Checks if the value of variable1 is less than or equal to variable2. | Syntax:   1. 1 2. variable1 <= variable2   Example 1:   1. 5 <= 5 and 3 <= 5   returns True  Example 2:   1. size = 38 2. max\_size = 40 3. size <= max\_size   returns True |
| Less Than(<) | Checks if the value of variable1 is less than variable2. | Syntax:   1. variable1 < variable2   Example 1:   1. 4 < 6   Copied!  returns True  Example 2:   1. score = 60 2. passing\_score = 65 3. score < passing\_score   returns True |
| Loop Controls | `break` exits the loop prematurely. `continue` skips the rest of the current iteration and moves to the next iteration. | Syntax:   1. for: # Code to repeat 2. if # boolean statement 3. break 4. for: # Code to repeat 5. if # boolean statement 6. continue   Example 1:   1. for num in range(1, 6): 2. if num == 3: 3. break 4. print(num)   Example 2:   1. for num in range(1, 6): 2. if num == 3: 3. continue 4. print(num) |
| NOT | Returns `True` if variable is `False`, and vice versa. | Syntax:   1. !variable   Example:   1. !isLocked   returns True if the variable is False (i.e., unlocked). |
| Not Equal(!=) | Checks if two values are not equal. | Syntax:   1. variable1 != variable2   Example:   1. a = 10 2. b = 20 3. a != b   returns True  Example 2:   1. count=0 2. count != 0   returns False |
| Object Creation | Creates an instance of a class (object) using the class constructor. | Syntax:   1. object\_name = ClassName(arguments)   Example:   1. person1 = Person("Alice", 25) |
| OR | Returns `True` if either statement1 or statement2 (or both) are `True`. Otherwise, returns `False`. | Syntax:   1. statement1 || statement2   Copied!  Example:   1. "Farewell Party Invitation" 2. Grade = 12 grade == 11 or grade == 12   returns True |
| range() | Generates a sequence of numbers within a specified range. | Syntax:   1. range(stop) 2. range(start, stop) 3. range(start, stop, step)   Example:   1. range(5) #generates a sequence of integers from 0 to 4. 2. range(2, 10) #generates a sequence of integers from 2 to 9. 3. range(1, 11, 2) #generates odd integers from 1 to 9. |
| Return Statement | `Return` is a keyword used to send a value back from a function to its caller. | Syntax:   1. return value   Example:   1. def add(a, b): return a + b 2. result = add(3, 5) |
| Try-Except Block | Tries to execute the code in the try block. If an exception of the specified type occurs, the code in the except block is executed. | Syntax:   1. try: # Code that might raise an exception except 2. ExceptionType: # Code to handle the exception   Example:   1. try: 2. num = int(input("Enter a number: ")) 3. except ValueError: 4. print("Invalid input. Please enter a valid number.") |
| Try-Except with Else Block | Code in the `else` block is executed if no exception occurs in the try block. | Syntax:   1. try: # Code that might raise an exception except 2. ExceptionType: # Code to handle the exception 3. else: # Code to execute if no exception occurs   Example:   1. try: 2. num = int(input("Enter a number: ")) 3. except ValueError: 4. print("Invalid input. Please enter a valid number") 5. else: 6. print("You entered:", num) |
| Try-Except with Finally Block | Code in the `finally` block always executes, regardless of whether an exception occurred. | Syntax:   1. try: # Code that might raise an exception except 2. ExceptionType: # Code to handle the exception 3. finally: # Code that always executes   Example:   1. try: 2. file = open("data.txt", "r") 3. data = file.read() 4. except FileNotFoundError: 5. print("File not found.") 6. finally: 7. file.close() |
| While Loop | A `while` loop repeatedly executes a block of code as long as a specified condition remains `True`. | Syntax:   1. while condition: # Code to repeat   Copied!  Example:   1. count = 0 while count < 5: 2. print(count) count += 1 |

| **Term** | **Definition** |
| --- | --- |
| Analogy | Refers to a concept or comparison outside the scope of the programming language itself, used to explain or relate one concept to another in a more understandable way. |
| Attributes | Attributes in Python refer to the characteristics or properties of an object, and they can be accessed using dot notation. |
| Branching | Branching in Python is a process of altering the flow of a program based on conditions, typically using if, elif, and else statements. |
| Comparison operators | Comparison operators in Python are used to compare values and return Boolean results (True or False), including operators like == (equal),!= (not equal), < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to). |
| Conditions | Conditions in Python are used to make decisions in code, executing specific blocks of code based on whether a given expression evaluates to True or False. |
| Enumerate | In Python, "enumerate" is a built-in function that adds a counter to an iterable, allowing you to loop through both the elements and their corresponding indices. |
| Exception handling | Exception handling in Python is a mechanism for gracefully managing and responding to errors or exceptional conditions that may occur during program execution. |
| Explicitly | In Python, the term "explicitly" refers to performing an action or specifying something in a clear, unambiguous, and direct manner. |
| For loops | For loops in Python are used for iterating over a sequence (such as a list, tuple, or string) or other iterable objects, executing a set of statements for each item in the sequence. |
| Global variable | Global variables in Python are variables defined outside of any function or block and can be accessed and modified from any part of the code. |
| Incremented | "Incremented" in Python means to increase the value of a variable by a specified amount, typically done using the += operator or by adding a fixed value. |
| Indent | In Python, "indent" refers to the use of whitespace at the beginning of a line to signify the structure and scope of code blocks, such as loops and functions. |
| Indices | In Python, "indices" refer to the position or location of elements in a sequence, like a string, list, or tuple, starting with 0 for the first element. |
| Iterate | In Python, "iterate" means to repeatedly perform a set of operations or steps on each item in a collection, such as a list, tuple, or dictionary, typically using loops or iterators. |
| Local variables | Local variables in Python are variables defined within a specific function or block of code and are only accessible within that function or block. |
| Logic operators | Logic operators in Python are used to perform logical operations on Boolean values, including operators like and (logical AND), or (logical OR), and not (logical NOT). |
| Loops | Loops in Python are constructs for repeating a block of code, enabling the execution of the same code multiple times. |
| Parameters | Parameters in Python are placeholders in a function definition, used to accept and work with values provided to the function when it is called. |
| Programming Fundamentals | Programming fundamentals in Python involve variables, control structures, functions, data structures, input/output, and error handling for building software. |
| Range function | The range function in Python generates a sequence of numbers that can be used for iterating in a loop and is typically used as range (start, stop, step), where it creates numbers from start to stop-1 with the given step increment. |
| Scope of function | The "scope of a function" in Python refers to the region of code where a variable defined within that function is accessible or visible. |
| Sequences | Sequences in Python are ordered collections of items that can include data types like strings, lists, and tuples, allowing for indexing and iteration. |
| Syntax | In Python, "explicitly" means to state something clearly and directly, leaving no room for ambiguity or implicit interpretation. |
| While loops | While loops in Python are used to repeatedly execute a block of code as long as a specified condition is true. |

**Writing to a file**

You can create a new text file and write data to it using Python's open() function. The open() function takes two main arguments: the file path (including the file name) and the mode parameter, which specifies the operation you want to perform on the file. For writing, you should use the mode 'w' Here's an example:

1. # Create a new file Example2.txt for writing
2. with open('Example2.txt', 'w') as file1:
3. file1.write("This is line A\n")
4. file1.write("This is line B\n")
5. # file1 is automatically closed when the 'with' block exits

Copied!

**Line 2 explanation:\*\* with open('Example2.txt', 'w') as file1:**

* We start by using the open function to open or create a file named Example2.txt for writing ('w' mode).
* The 'w' mode specifies that we intend to write data to the file.
* We use the with statement to ensure that the file is automatically closed when the code block exits. This helps manage resources efficiently.

**Line 3 explanation: file1.write("This is line A\n")**

* Here, we use the write() method on the file object, file1, to add the text This is line A to the file.
* The \n at the end represents a newline character, which starts a new line in the file.

**Line 4 explanation file1.write("This is line" B\n")**

* Similarly, we use the write() method again to add the text This is line B to the file on a new line.

**Writing multiple lines to a file using a list and loop**

In Python, you can use a list to store multiple lines of text and then write these lines to a file using a loop. Here's an example code snippet that demonstrates this:

1. # List of lines to write to the file
2. Lines = ["This is line 1", "This is line 2", "This is line 3"]
3. # Create a new file Example3.txt for writing
4. with open('Example3.txt', 'w') as file2:
5. for line in Lines:
6. file2.write(line + "\n")
7. # file2 is automatically closed when the 'with' block exits

Here's an explanation of the code:

* Line 2: We start by defining a list called Lines, which contains multiple lines of text that we want to write to the file. Each line is a string.
* Line 5: Next, we use the open() function to create a new text file named Example3.txt for writing, 'w' mode. The 'w' mode indicates that we intend to write data to the file.
* Line 6: We then enter a for loop to iterate through each element (line) in the Lines list.
* Line 7: Inside the loop, we use the write() method on the file object file2 to write the current line of text (line) to the file. We add \n at the end of each line to ensure that each line is followed by a newline character, which separates them in the file.
* Line 8: Finally, we add a comment indicating that the file file2 will be automatically closed when the code block within the with statement exits. Properly closing the file is essential for good resource management.

**Appending data to an existing file**

In Python, you can use the 'a' mode when opening a file to append new data to an existing file without overwriting its contents. Here's an example code snippet that demonstrates this:

1. # Data to append to the existing file
2. new\_data = "This is line C"
3. # Open an existing file Example2.txt for appending
4. with open('Example2.txt', 'a') as file1:
5. file1.write(new\_data + "\n")
6. # file1 is automatically closed when the 'with' block exits

Here's an explanation of the code:

* Line 2: We start by defining a variable new\_data that contains the text we want to append to the existing file. In this case, it's the string `This is line C.``
* Line 5: Next, we use the open() function to open an existing file named Example2.txt for appending, 'a' mode. The 'a' mode indicates that we intend to append data to the file, and if the file doesn't exist, it will be created.
* Line 6: Within the with block, we use the write() method on the file object file1 to append the new\_data to the file. We add "\n" at the end to ensure that the appended data starts on a new line, maintaining the file's readability.
* Finally, we add a comment indicating that the file file1 will automatically close when the code block within the with statement exits. Properly closing the file is essential for good resource management.

**Copying contents from one file to another**

In Python, you can copy the contents of one file to another by reading from the source file and writing to the destination file. Here's an example code snippet that demonstrates this:

1. # Open the source file for reading
2. with open('source.txt', 'r') as source\_file:
3. # Open the destination file for writing
4. with open('destination.txt', 'w') as destination\_file:
5. # Read lines from the source file and copy them to the destination file
6. for line in source\_file:
7. destination\_file.write(line)
8. # Destination file is automatically closed when the 'with' block exits
9. # Source file is automatically closed when the 'with' block exits

Copied!

Here's an explanation of the code:

* Line 2: We start by opening the source file, source.txt for reading, r mode, using the with statement and the open() function. This allows us to read data from the source file.
* Line 4: Inside the first with block, we open the destination file, destination.txt for writing, w mode, using another with statement and the open() function. This prepares the destination file for writing.
* Line 6: We use a for loop to iterate through each line in the source file source\_file. This loop reads each line from the source file one by one.
* Line 7: Within the loop, we use the write() method to write each line from the source file to the destination file destination\_file. This effectively copies the content of the source file to the destination file.
* Lines 8 and 9: After copying all the lines, both the source and destination files are automatically closed when their respective with blocks exit. Proper file closure is crucial for managing resources efficiently.

**File modes in Python (syntax and use cases)**

The following table provides an overview of different file modes, their syntax, and common use cases. Understanding these modes is essential when working with files in Python for various data manipulation tasks.

| **Mode** | **Syntax** | **Description** |
| --- | --- | --- |
| ‘r’ | 'r' | Read mode. Opens an existing file for reading. Raises an error if the file doesn't exist. |
| ‘w’ | 'w' | Write mode. Creates a new file for writing. Overwrites the file if it already exists. |
| ‘a’ | 'a' | Append mode. Opens a file for appending data. Creates the file if it doesn't exist. |
| ‘x’ | 'x' | Exclusive creation mode. Creates a new file for writing but raises an error if the file already exists. |
| ‘rb’ | 'rb' | Read binary mode. Opens an existing binary file for reading. |
| ‘wb’ | 'wb' | Write binary mode. Creates a new binary file for writing. |
| ‘ab’ | 'ab' | Append binary mode. Opens a binary file for appending data. |
| ‘xb’ | 'xb' | Exclusive binary creation mode. Creates a new binary file for writing but raises an error if it already exists. |
| ‘rt’ | 'rt' | Read text mode. Opens an existing text file for reading. (Default for text files) |
| ‘wt’ | 'wt' | Write text mode. Creates a new text file for writing. (Default for text files) |
| ‘at’ | 'at' | Append text mode. Opens a text file for appending data. (Default for text files) |
| ‘xt’ | 'xt' | Exclusive text creation mode. Creates a new text file for writing but raises an error if it already exists. |
| ‘r+’ | 'r+' | Read and write mode. Opens an existing file for both reading and writing. |
| ‘w+’ | 'w+' | Write and read mode. Creates a new file for reading and writing. Overwrites the file if it already exists. |
| ‘a+’ | 'a+' | Append and read mode. Opens a file for both appending and reading. Creates the file if it doesn't exist. |
| ‘x+’ | 'x+' | Exclusive creation and read/write mode. Creates a new file for reading and writing but raises an error if it already exists. |

**What is Pandas?**

Pandas is a popular open-source data manipulation and analysis library for the Python programming language. It provides a powerful and flexible set of tools for working with structured data, making it a fundamental tool for data scientists, analysts, and engineers.  
Pandas is designed to handle data in various formats, such as tabular data, time series data, and more, making it an essential part of the data processing workflow in many industries.

Here are some **key features and functionalities of Pandas**:

**Data Structures**: Pandas offers two primary data structures - DataFrame and Series.

1. A DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).
2. A Series is a one-dimensional labeled array, essentially a single column or row of data.

**Data Import and Export**: Pandas makes it easy to read data from various sources, including CSV files, Excel spreadsheets, SQL databases, and more. It can also export data to these formats, enabling seamless data exchange.

**Data Merging and Joining**: You can combine multiple DataFrames using methods like merge and join, similar to SQL operations, to create more complex datasets from different sources.

**Efficient Indexing**: Pandas provides efficient indexing and selection methods, allowing you to access specific rows and columns of data quickly.

**Custom Data Structures**: You can create custom data structures and manipulate data in ways that suit your specific needs, extending Pandas' capabilities.

**Importing Pandas:**

Import Pandas using the import command, followed by the library's name.  
Commonly, Pandas is imported as pd for brevity in code.

1. import pandas as pd

**Data Loading:**

* Pandas can be used to load data from various sources, such as CSV and Excel files.
* The read\_csv function is used to load data from a CSV file into a Pandas DataFrame.

To read a CSV (Comma-Separated Values) file in Python using the Pandas library, you can use the pd.read\_csv() function. Here's the syntax to read a CSV file:

1. import pandas as pd
2. # Read the CSV file into a DataFrame
3. df = pd.read\_csv('your\_file.csv')

Replace 'your\_file.csv' with the actual file path of your CSV file. Make sure that the file is located in the same directory as your Python script, or you provide the correct file path.

**What is a Series?**

A Series is a one-dimensional labeled array in Pandas. It can be thought of as a single column of data with labels or indices for each element. You can create a Series from various data sources, such as lists, NumPy arrays, or dictionaries  
Here's a basic example of creating a Series in Pandas:

1. import pandas as pd
2. # Create a Series from a list
3. data = [10, 20, 30, 40, 50]
4. s = pd.Series(data)
5. print(s)

In this example, we've created a Series named **s** with numeric data. Notice that Pandas automatically assigned numerical indices (0, 1, 2, 3, 4) to each element, but you can also specify custom labels if needed.

**Accessing Elements in a Series**

You can access elements in a Series using the index labels or integer positions. Here are a few common methods for accessing Series data:

**Accessing by label**

1. print(s[2]) # Access the element with label 2 (value 30)

**Accessing by position**

1. print(s.iloc[3]) # Access the element at position 3 (value 40)

**Accessing multiple elements**

1. print(s[1:4]) # Access a range of elements by label

**Series Attributes and Methods**

Pandas Series come with various attributes and methods to help you manipulate and analyze data effectively. Here are a few essential ones:

* **values**: Returns the Series data as a NumPy array.
* **index**: Returns the index (labels) of the Series.
* **shape**: Returns a tuple representing the dimensions of the Series.
* **size**: Returns the number of elements in the Series.
* **mean(), sum(), min(), max()**: Calculate summary statistics of the data.
* **unique(), nunique()**: Get unique values or the number of unique values.
* **sort\_values(), sort\_index()**: Sort the Series by values or index labels.
* **isnull(), notnull()**: Check for missing (NaN) or non-missing values.
* **apply()**: Apply a custom function to each element of the Series.

**What is a DataFrames?**

A DataFrame is a two-dimensional labeled data structure with columns of potentially different data types. Think of it as a table where each column represents a variable, and each row represents an observation or data point. DataFrames are suitable for a wide range of data, including structured data from CSV files, Excel spreadsheets, SQL databases, and more.

**Creating DataFrames from Dictionaries:**

DataFrames can be created from dictionaries, with keys as column labels and values as lists representing rows.

1. import pandas as pd
2. # Creating a DataFrame from a dictionary
3. data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
4. 'Age': [25, 30, 35, 28],
5. 'City': ['New York', 'San Francisco', 'Los Angeles', 'Chicago']}
6. df = pd.DataFrame(data)
7. print(df)

**Column Selection:**

You can select a single column from a DataFrame by specifying the column name within double brackets.  
Multiple columns can be selected in a similar manner, creating a new DataFrame.

1. print(df['Name']) # Access the 'Name' column

**Accessing Rows:**

You can access rows by their index using .iloc[] or by label using .loc[].

1. print(df.iloc[2]) # Access the third row by position
2. print(df.loc[1]) # Access the second row by label

**Slicing:**

You can slice DataFrames to select specific rows and columns.

1. print(df[['Name', 'Age']]) # Select specific columns
2. print(df[1:3]) # Select specific rows

**Finding Unique Elements:**

Use the unique method to determine the unique elements in a column of a DataFrame.

1. unique\_dates = df['Age'].unique()

**Conditional Filtering:**

You can filter data in a DataFrame based on conditions using inequality operators.  
For instance, you can filter albums released after a certain year.

1. high\_above\_102 = df[df['Age'] > 25]

**Saving DataFrames:**

To save a DataFrame to a CSV file, use the to\_csv method and specify the filename with a “.csv” extension.Pandas provides other functions for saving DataFrames in different formats.

1. df.to\_csv('trading\_data.csv', index=False)

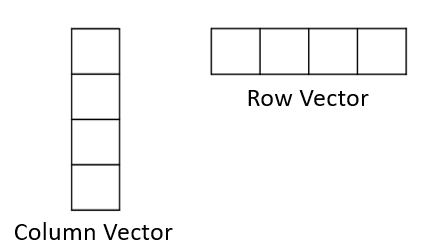
**DataFrame Attributes and Methods**

DataFrames provide numerous attributes and methods for data manipulation and analysis, including:

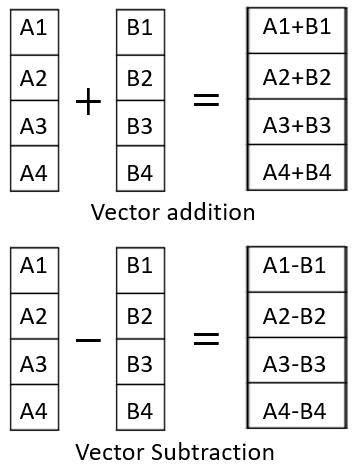
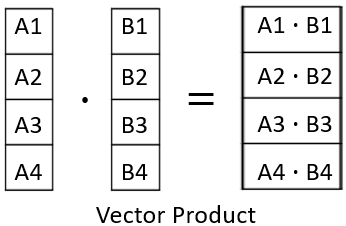
* **shape**: Returns the dimensions (number of rows and columns) of the DataFrame.
* **info()**: Provides a summary of the DataFrame, including data types and non-null counts.
* **describe()**: Generates summary statistics for numerical columns.
* **head(), tail()**: Displays the first or last n rows of the DataFrame.
* **mean(), sum(), min(), max()**: Calculate summary statistics for columns.
* **sort\_values()**: Sort the DataFrame by one or more columns.
* **groupby()**: Group data based on specific columns for aggregation.
* **fillna(), drop(), rename()**: Handle missing values, drop columns, or rename columns.
* **apply()**: Apply a function to each element, row, or column of the DataFrame.

*Pandas offers a wide range of methods beyond these examples. For more detailed information, please refer to the official documentation available on the*[*Pandas official website*](https://pandas.pydata.org/docs/)*.*

**1D Arrays : Vectors**

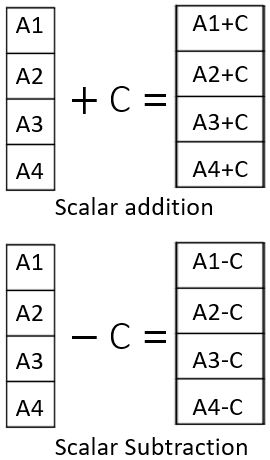
A 1D array is often termed as a vector. Depending upon the orientation of the data, the vector can be classified as a row vector or a column vector. This is illustrated in the image below.  


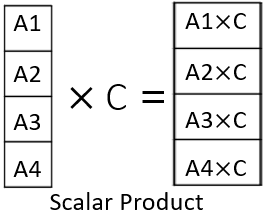
Mathematically, we can add, subtract, and take the product of two vectors, provided they are the same shape. The images below highlight the mathematical operations conducted on a pair of vectors.

All three of these operations are conducted on corresponding elements of individual vectors. The resulting array always has the same size as that of the two original vectors.

To a single vector, we can also add a constant (scalar addition), subtract a constant (scalar subtraction) and multiply a constant (scalar multiplication) to any vector. The images below illustrate these operations.



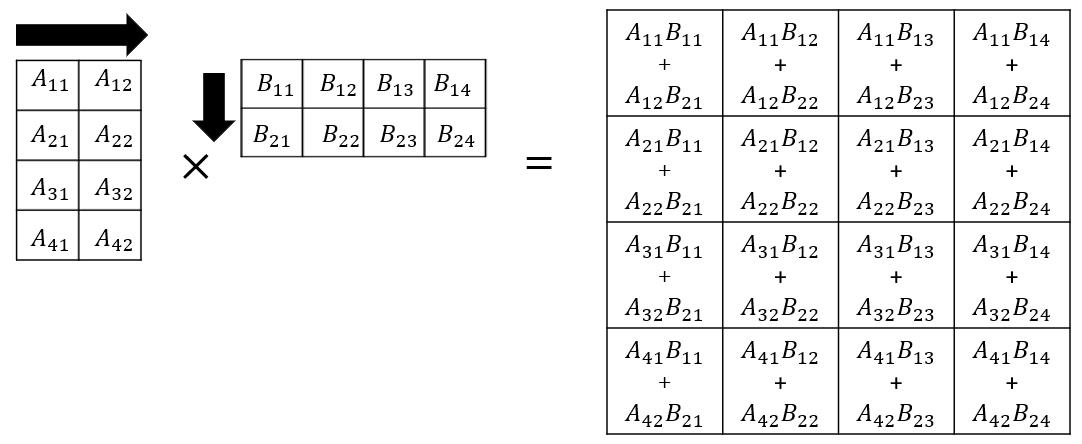


**2D Arrays : Matrices**

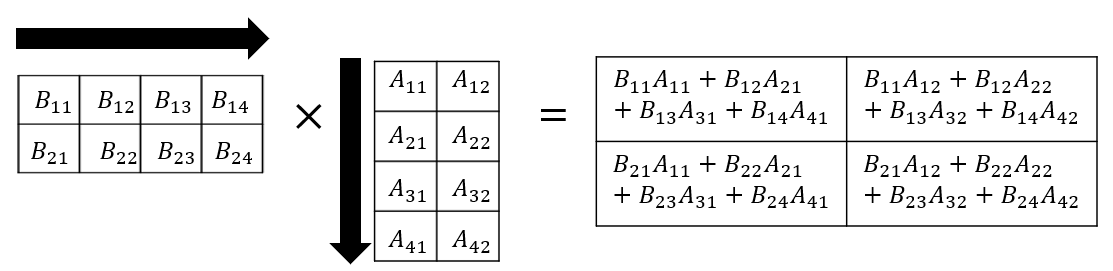
A 2D array is also called a Matrix. These are typically rectangular arrays with data stored in different rows. All of the operations mentioned above are also applicable to the 2D arrays. However, the Dot product of 2D matrices follows a different rule.

As illlustrated in the images below, the dot product is carried out by multiplying and adding corresponding elements of rows of the first matrix with the elements of columns of the second matrix. As a result, the output matrix from the multiplication will have a modified shape.

The general rule is that the dot product of an m X n matrix can be done only with an n X p matrix, and the resultant matrix will have the shape m X p. In the example shown below, the 4 X 2 matrix is multiplied with the 2 X 4 matrix to generate a 4 X 4 matrix.



In the reverse example, when 2 X 4 matrix is multiplied with the 4 X 2 one, the resultant will be a 2 X 2 matrix.



*Note: Dot product of a row vector with a column vector, with the same number of elements, would return a single scalar value. Dot product of a column vector with a row vector, will return a 2D matrix.*

**What is NumPy?**

NumPy, short for **Num**erical **Py**thon, is a fundamental library for numerical and scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays. NumPy serves as the foundation for many data science and machine learning libraries, making it an essential tool for data analysis and scientific research in Python.

**Key aspects of NumPy in Python:**

* **Efficient data structures**: NumPy introduces efficient array structures, which are faster and more memory-efficient than Python lists. This is crucial for handling large data sets.
* **Multi-dimensional arrays**: NumPy allows you to work with multi-dimensional arrays, enabling the representation of matrices and tensors. This is particularly useful in scientific computing.
* **Element-wise operations**: NumPy simplifies element-wise mathematical operations on arrays, making it easy to perform calculations on entire data sets in one go.
* **Random number generation**: It provides a wide range of functions for generating random numbers and random data, which is useful for simulations and statistical analysis.
* **Integration with other libraries**: NumPy seamlessly integrates with other data science libraries like SciPy, Pandas, and Matplotlib, enhancing its utility in various domains.
* **Performance optimization**: NumPy functions are implemented in low-level languages like C and Fortran, which significantly boosts their performance. It's a go-to choice when speed is essential.

**Installation**

If you haven't already installed NumPy, you can do so using pip:

1. pip install numpy

**Creating NumPy arrays**

You can create NumPy arrays from Python lists. These arrays can be one-dimensional or multi-dimensional.

**Creating 1D array**

1. import numpy as np

**import numpy as np**: In this line, the NumPy library is imported and assigned an alias np to make it easier to reference in the code.

1. # Creating a 1D array
2. arr\_1d = np.array([1, 2, 3, 4, 5]) # \*\*np.array()\*\* is used to create NumPy arrays.

**arr\_1d = np.array([1, 2, 3, 4, 5])**: In this line, a one-dimensional NumPy array named arr\_1d is created. It uses the np.array() function to convert a Python list [1, 2, 3, 4, 5] into a NumPy array. This array contains five elements, which are 1, 2, 3, 4, and 5. arr\_1d is a 1D array because it has a single row of elements.

**Creating 2D array**

1. import numpy as np

**import numpy as np**: In this line, the NumPy library is imported and assigned an alias np to make it easier to reference in the code.

1. # Creating a 2D array
2. arr\_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

**arr\_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])**: In this line, a two-dimensional NumPy array named arr\_2d is created. It uses the np.array() function to convert a list of lists into a 2D NumPy array.  
The outer list contains three inner lists, each of which represents a row of elements. So, arr\_2d is a 2D array with three rows and three columns. The elements in this array form a matrix with values from 1 to 9, organized in a 3x3 grid.

**Array attributes**

NumPy arrays have several useful attributes:

1. # Array attributes
2. print(arr\_2d.ndim) # ndim : Represents the number of dimensions or "rank" of the array.
3. # output : 2
4. print(arr\_2d.shape) # shape : Returns a tuple indicating the number of rows and columns in the array.
5. # Output : (3, 3)
6. print(arr\_2d.size) # size: Provides the total number of elements in the array.
7. # Output : 9

**Indexing and slicing**

You can access elements of a NumPy array using indexing and slicing:

In this line, the third element (index 2) of the 1D array arr\_1d is accessed.

1. # Indexing and slicing
2. print(arr\_1d[2]) # Accessing an element (3rd element)

In this line, the element in the 2nd row (index 1) and 3rd column (index 2) of the 2D array arr\_2d is accessed.

1. print(arr\_2d[1, 2]) # Accessing an element (2nd row, 3rd column)

In this line, the 2nd row (index 1) of the 2D array arr\_2d is accessed.

1. print(arr\_2d[1]) # Accessing a row (2nd row)

In this line, the 2nd column (index 1) of the 2D array arr\_2d is accessed.

1. print(arr\_2d[:, 1]) # Accessing a column (2nd column)

**Basic operations**

NumPy simplifies basic operations on arrays:

**Element-wise arithmetic operations:**

Addition, subtraction, multiplication, and division of arrays with scalars or other arrays.

**Array addition**

1. # Array addition
2. array1 = np.array([1, 2, 3])
3. array2 = np.array([4, 5, 6])
4. result = array1 + array2
5. print(result) # [5 7 9]

**Scalar multiplication**

1. # Scalar multiplication
2. array = np.array([1, 2, 3])
3. result = array \* 2 # each element of an array is multiplied by 2
4. print(result) # [2 4 6]

**Element-wise multiplication (Hadamard Product)**

1. # Element-wise multiplication (Hadamard product)
2. array1 = np.array([1, 2, 3])
3. array2 = np.array([4, 5, 6])
4. result = array1 \* array2
5. print(result) # [4 10 18]

**Matrix multiplication**

1. # Matrix multiplication
2. matrix1 = np.array([[1, 2], [3, 4]])
3. matrix2 = np.array([[5, 6], [7, 8]])
4. result = np.dot(matrix1, matrix2)
5. print(result)
6. # [[19 22]
7. # [43 50]]

NumPy simplifies these operations, making it easier and more efficient than traditional Python lists.

**Operation with NumPy**

Here's the list of operation which can be performed using Numpy

| **Operation** | **Description** | **Example** |
| --- | --- | --- |
| Array Creation | Creating a NumPy array. | arr = np.array([1, 2, 3, 4, 5]) |
| Element-Wise Arithmetic | Element-wise addition, subtraction, and so on. | result = arr1 + arr2 |
| Scalar Arithmetic | Scalar addition, subtraction, and so on. | result = arr \* 2 |
| Element-Wise Functions | Applying functions to each element. | result = np.sqrt(arr) |
| Sum and Mean | Calculating the sum and mean of an array.Calculating the sum and mean of an array. | total = np.sum(arr)<br>average = np.mean(arr) |
| Maximum and Minimum Values | Finding the maximum and minimum values. | max\_val = np.max(arr)<br>min\_val = np.min(arr) |
| Reshaping | Changing the shape of an array. | reshaped\_arr = arr.reshape(2, 3) |
| Transposition | Transposing a multi-dimensional array. | transposed\_arr = arr.T |
| Matrix Multiplication | Performing matrix multiplication. | result = np.dot(matrix1, matrix2) |

**What are APIs?**

APIs, or Application Programming Interfaces, are a crucial part of software development. They allow developers to create new applications by leveraging existing functionality from other systems. APIs define how software components should interact and facilitate communication between various products and services without requiring direct implementation.

**Importance of APIs**

APIs are essential for any engineer because they provide a way to access data and functionality from other systems, which can save time and resources. For instance, APIs can be used to integrate applications into the existing architecture of a server or application, allowing developers to communicate between various products and services without requiring direct implementation.

APIs are also important because they enable developers to create new applications by leveraging existing functionality from other systems. This can help developers throughout the engineering and development process of apps.

APIs are used in a wide range of applications, from social media platforms to e-commerce websites. They are also used in mobile applications, web applications, and desktop applications.

**Applications of APIs**

APIs have a wide range of applications, some of which are:

1. **Social media platforms:** Social media platforms like Facebook, Twitter, and Instagram use APIs to allow developers to access their data and functionality. This allows developers to create applications that can interact with these platforms and provide additional functionality to users.
2. **E-commerce websites:** E-commerce websites like Amazon and eBay use APIs to allow developers to access their product catalogs and other data. This allows developers to create applications that can interact with these platforms and provide additional functionality to users.
3. **Weather applications:** Weather applications like AccuWeather and The Weather Channel use APIs to access weather data from various sources. This allows developers to create applications that can provide users with up-to-date weather information.
4. **Maps and navigation applications:** Maps and navigation applications like Google Maps and Waze use APIs to access location data and other information. This allows developers to create applications that can provide users with directions, traffic updates, and other location-based information.
5. **Payment gateways:** Payment gateways like PayPal and Stripe use APIs to allow developers to access their payment processing functionality. This allows developers to create applications that can process payments securely and efficiently.
6. **Messaging applications:** Messaging applications like WhatsApp and Facebook Messenger use APIs to allow developers to access their messaging functionality. This allows developers to create applications that can interact with these platforms and provide additional functionality to users.

* Python uses the open() function and allows you to read and write files, providing access to the content within the file for reading. It also allows overwriting it for writing and specifies the file mode (for example, r for reading, w for writing, a for appending).
  + To read a file, Python uses an open function along with *r.*
  + Python uses the**open with** function to read and process a file attribute, that is, from open to close.
  + In Python, you use the **open** method to edit or overwrite a file.
  + To write a file, Python uses the **open** function along with *w.*
  + In Python, "a" indicates that the program has appended to the file.
  + In Python, “\n” signifies that the code should start on a new line.
  + Python uses various methods to print lines from attributes.
* Pandas is a powerful Python library for data manipulation and analysis, providing data structures and functions to work with structured data like data frames and series.
  + You import the file (panda) by using the import command followed by the file name.
  + In Python, you use the **as** command to provide a shorter name for the file.
  + In Pandas, you use a data frame (df) to specify the files to read.
  + DataFrames consist of rows and columns.
  + You can create new DataFrames by using the column or columns of a specific DataFrame.
  + We can work with data in a DataFrames and save the results in different formats.
  + In Python, you use the **Unique** method to determine unique elements in a column of the DataFrames.
  + You use the inequality operator along with df to assign a Boolean value to the selected column in DataFrames.
  + You save a new DataFrame as a different DataFrame, which may contain values from an earlier DataFrame.
* NumPy is a Python library for numerical and matrix operations, offering multidimensional array objects and a variety of mathematical functions to work with data efficiently.
  + NumPy is a basis for Pandas.
  + A NumPy array or ND array is similar to a list, usually of a fixed size with the same kind of element.
* A one-dimensional NumPy array is a linear sequence of elements with a single axis, like a traditional list, but optimized for numerical computations and array operations.
  + You can access elements in a NumPy using an index.
  + You use the attribute **dtype** to get the data type of the array elements.
  + You use **nsize** and **ndim** to get the size and dimension of the array, respectively.
  + You can use indexing and slicing methods in NumPy.
  + Vector additions are widely used operations in Python.
  + Representing vector addition with line segments or arrows is useful.
  + NumPy codes work much faster, which is helpful with lots of data.
  + You perform vector subtraction by replacing the addition sign with a negative sign.
  + Multiplying an array by a scalar in Python entails multiplying each element of the array by the scalar value, leading to a new array in which each element scales by the scalar.
  + Hadamard product refers to the element-wise multiplication of two arrays of the same shape, resulting in a new array where each element is the product of the corresponding elements in the input arrays.
  + The dot product in Python is the sum of the element-wise products of two arrays, often used for vector and matrix operations to find the scalar result of multiplying corresponding elements and summing them.
  + When working with NumPy, it is common to utilize libraries like Matplotlib to create graphs and visualizations from numerical data stored in NumPy arrays.
* A two-dimensional NumPy array is a grid-like structure with rows and columns suitable for representing data as a matrix or a table for numerical computations.
  + In NumPy, "shape" refers to an array's dimensions (number of rows and columns), indicating its size and structure.
  + You use the attribute "size" to obtain the size of an array.
  + You use rectangular attributes to access the various elements in an array.
  + You use a scalar to multiply elements in NumPy.

**Working with Data in Python Cheat Sheet**

**Reading and writing files**

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Syntax and Code Example** |
| File opening modes | Different modes to open files for specific operations. | Syntax: r (reading) w (writing) a (appending) + (updating: read/write) b (binary, otherwise text)   1. Examples: with open("data.txt", "r") as file: content = file.read() print(content) with open("output.txt", "w") as file: file.write("Hello, world!") with open("log.txt", "a") as file: file.write("Log entry: Something happened.") with open("data.txt", "r+") as file: content = file.read() file.write("Updated content: " + content)</td> |
| File reading methods | Different methods to read file content in various ways. | Syntax:   1. file.readlines() # reads all lines as a list 2. readline() # reads the next line as a string 3. file.read() # reads the entire file content as a string   Example:   1. with open("data.txt", "r") as file: 2. lines = file.readlines() 3. next\_line = file.readline() 4. content = file.read() |
| File writing methods | Different write methods to write content to a file. | Syntax:   1. file.write(content) # writes a string to the file 2. file.writelines(lines) # writes a list of strings to the file   Example:   1. lines = ["Hello\n", "World\n"] 2. with open("output.txt", "w") as file: 3. file.writelines(lines) |
| Iterating over lines | Iterates through each line in the file using a `loop`. | Syntax:   1. for line in file: # Code to process each line   Example:   1. with open("data.txt", "r") as file: 2. for line in file: print(line) |
| Open() and close() | Opens a file, performs operations, and explicitly closes the file using the close() method. | Syntax:   1. file = open(filename, mode) # Code that uses the file 2. file.close()   Example:   1. file = open("data.txt", "r") 2. content = file.read() 3. file.close() |
| with open() | Opens a file using a with block, ensuring automatic file closure after usage. | Syntax:   1. with open(filename, mode) as file: # Code that uses the file   Example:   1. with open("data.txt", "r") as file: 2. content = file.read() |

**Pandas**

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Syntax and Code Example** |
| .read\_csv() | Reads data from a `.CSV` file and creates a DataFrame. | Syntax: dataframe\_name = pd.read\_csv("filename.csv") Example: df = pd.read\_csv("data.csv") |
| .read\_excel() | Reads data from an Excel file and creates a DataFrame. | Syntax:   1. dataframe\_name = pd.read\_excel("filename.xlsx")   Example:   1. df = pd.read\_excel("data.xlsx") |
| .to\_csv() | Writes DataFrame to a CSV file. | Syntax:   1. dataframe\_name.to\_csv("output.csv", index=False)   Example:   1. df.to\_csv("output.csv", index=False) |
| Access Columns | Accesses a specific column using [] in the DataFrame. | Syntax:   1. dataframe\_name["column\_name"] # Accesses single column 2. dataframe\_name[["column1", "column2"]] # Accesses multiple columns   Example:   1. df["age"] 2. df[["name", "age"]] |
| describe() | Generates statistics summary of numeric columns in the DataFrame. | Syntax:   1. dataframe\_name.describe()   Example:   1. df.describe() |
| drop() | Removes specified rows or columns from the DataFrame. axis=1 indicates columns. axis=0 indicates rows. | Syntax:   1. dataframe\_name.drop(["column1", "column2"], axis=1, inplace=True) 2. dataframe\_name.drop(index=[row1, row2], axis=0, inplace=True)   Example:   1. df.drop(["age", "salary"], axis=1, inplace=True) # Will drop columns 2. df.drop(index=[5, 10], axis=0, inplace=True) # Will drop rows |
| dropna() | Removes rows with missing NaN values from the DataFrame. axis=0 indicates rows. | Syntax:   1. dataframe\_name.dropna(axis=0, inplace=True)   Example:   1. df.dropna(axis=0, inplace=True) |
| duplicated() | Duplicate or repetitive values or records within a data set. | Syntax:   1. dataframe\_name.duplicated()   Example:   1. duplicate\_rows = df[df.duplicated()] |
| Filter Rows | Creates a new DataFrame with rows that meet specified conditions. | Syntax:   1. filtered\_df = dataframe\_name[(Conditional\_statements)]   Example:   1. filtered\_df = df[(df["age"] > 30) & (df["salary"] < 50000) |
| groupby() | Splits a DataFrame into groups based on specified criteria, enabling subsequent aggregation, transformation, or analysis within each group. | Syntax:   1. grouped = dataframe\_name.groupby(by, axis=0, level=None, as\_index=True, 2. sort=True, group\_keys=True, squeeze=False, observed=False, dropna=True)   Example:   1. grouped = df.groupby(["category", "region"]).agg({"sales": "sum"}) |
| head() | Displays the first n rows of the DataFrame. | Syntax:   1. dataframe\_name.head(n)   Example:   1. df.head(5) |
| Import pandas | Imports the Pandas library with the alias pd. | Syntax:   1. import pandas as pd   Example:   1. import pandas as pd |
| info() | Provides information about the DataFrame, including data types and memory usage. | Syntax:   1. dataframe\_name.info()   Example:   1. df.info() |
| merge() | Merges two DataFrames based on multiple common columns. | Syntax:   1. merged\_df = pd.merge(df1, df2, on=["column1", "column2"])   Example:   1. merged\_df = pd.merge(sales, products, on=["product\_id", "category\_id"]) |
| print DataFrame | Displays the content of the DataFrame. | Syntax:   1. 1 2. print(df) # or just type df   Copied!  Example:   1. print(df) 2. df |
| replace() | Replaces specific values in a column with new values. | Syntax:   1. dataframe\_name["column\_name"].replace(old\_value, new\_value, inplace=True)   Example:   1. df["status"].replace("In Progress", "Active", inplace=True) |
| tail() | Displays the last n rows of the DataFrame. | Syntax:   1. dataframe\_name.tail(n)   Example:   1. df.tail(5) |

**Numpy**

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Syntax and Code Example** |
| Importing NumPy | Imports the NumPy library. | Syntax:   1. import numpy as np   Example:   1. import numpy as np |
| np.array() | Creates a one or multi-dimensional array, | Syntax:   1. array\_1d = np.array([list1 values]) # 1D Array 2. array\_2d = np.array([[list1 values], [list2 values]]) # 2D Array   Example:   1. array\_1d = np.array([1, 2, 3]) # 1D Array 2. array\_2d = np.array([[1, 2], [3, 4]]) # 2D Array |
| Numpy Array Attributes | - Calculates the mean of array elements - Calculates the sum of array elements - Finds the minimum value in the array - Finds the maximum value in the array - Computes dot product of two arrays | Example:   1. np.mean(array) 2. np.sum(array) 3. np.min(array 4. np.max(array) 5. np.dot(array\_1, array\_2) |

Welcome! This alphabetized glossary contains many of the terms you'll find within this course. This comprehensive glossary also includes additional industry-recognized terms not used in course videos. These terms are important for you to recognize when working in the industry, participating in user groups, and participating in other certificate programs.

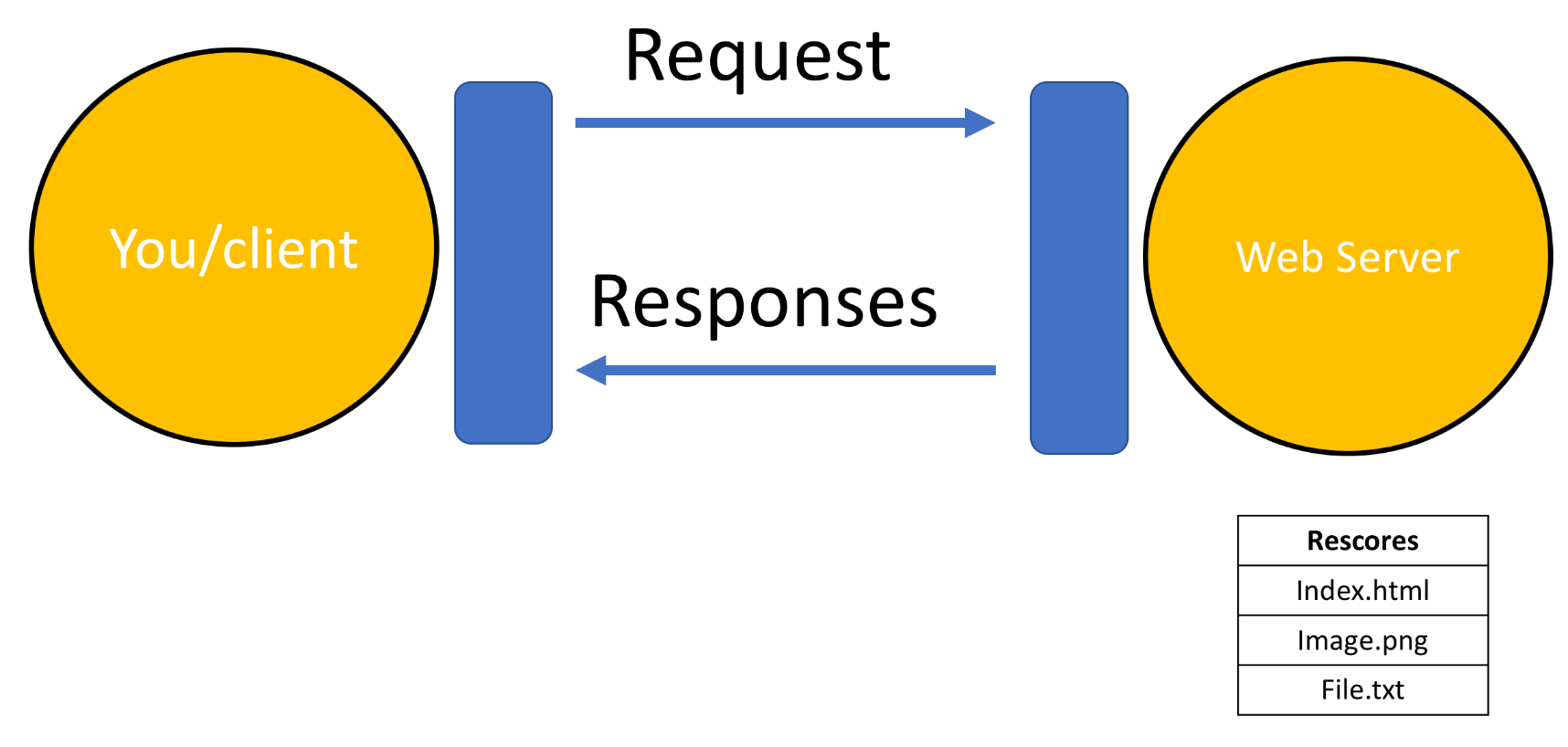
| **Term** | **Definition** |
| --- | --- |
| .csv file | A .csv (Comma-Separated Values) file is a plain text file format for storing tabular data, where each line represents a row and uses commas to separate values in different columns. |
| .txt file | A .txt (Text) file is a common file format that contains plain text without specific formatting, making it suitable for storing and editing textual data. |
| Append | To "append" means to add or attach something to the end of an existing object, typically used in the context of adding data to a file or elements to a data structure like a list in Python. |
| Attribute | An "attribute" in Python refers to a property or characteristic associated with an object, which can be accessed using dot notation. |
| Broadcasting in NumPy | Broadcasting in NumPy allows arrays with different shapes to be combined in element-wise operations by automatically extending smaller arrays to match the shape of larger ones, making operations more flexible. |
| Component | In NumPy, a "component" typically refers to a specific element or value within a multi-dimensional array, which can be accessed using indexing. |
| Computation | Computation in NumPy involves performing numerical operations on arrays and matrices, making it a powerful library for mathematical and scientific computing in Python. |
| Data analysis | Data analysis is the process of inspecting, cleaning, transforming, and interpreting data to discover useful information, draw conclusions, and support decision-making. |
| DataFrames | A DataFrames in Pandas is a two-dimensional, tabular data structure for storing and analyzing data, consisting of rows and columns. |
| Dependencies | Dependencies in Pandas are external libraries or modules, such as NumPy, that Pandas rely on for fundamental data manipulation and analysis functionality. |
| File attribute | File attributes generally refer to properties or metadata associated with files, which are managed at the operating system level. |
| File object | A "file object" in Python represents an open file, allowing reading from or writing to the file. |
| Grid | In Python, a "grid" typically refers to a two-dimensional structure composed of rows and columns, often used to represent data in a tabular format or for organizing objects in a coordinate system. |
| Hadamard Product | The Hadamard product is a mathematical operation that involves element-wise multiplication of two matrices or arrays of the same shape, producing a new matrix with each element being the product of the corresponding elements in the input matrices. |
| Importing pandas | To import Pandas in Python, you use the statement: import pandas as pd, which allows you to access Pandas functions and data structures using the abbreviation "pd." |
| Index | In Python, an "index" typically refers to a position or identifier used to access elements within a sequence or data structure, such as a list or string. |
| Libraries | Libraries in Python are collections of pre-written code modules that provide reusable functions and classes to simplify and enhance software development. |
| Linespace | In Python, "linespace" refers to a NumPy function that generates an array of evenly spaced values within a specified range. |
| NumPy | NumPy in Python is a fundamental library for numerical computing that provides support for large, multi-dimensional arrays and matrices, as well as a variety of high-level mathematical functions to operate on these arrays. |
| One dimensional NumPy | A one-dimensional NumPy array is a linear data structure that stores elements in a single sequence, often used for numerical computations and data manipulation. |
| Open function | In Python, the "open" function is used to access and manipulate files, allowing you to read from or write to a specified file. |
| Pandas | Pandas is a popular Python library for data manipulation and analysis, offering data structures and tools for working with structured data like tables and time series. |
| Pandas library | Pandas library in Python refer to the various modules and functions within the Pandas library, which provides powerful data structures and data analysis tools for working with structured data. |
| Plotting Mathematical Functions | Plotting mathematical functions in Python involves using libraries like Matplotlib to create graphical representations of mathematical equations, aiding visualization, and analysis. |
| Shape | In NumPy, "shape" refers to an array's dimensions (number of rows and columns), describing its size and structure. |
| Slicing | Slicing in NumPy entails extracting specific portions of an array by specifying a range of indices, enabling you to work with subsets of the data. |
| Two dimensional NumPy | A two-dimensional NumPy array is a structured data representation with rows and columns, resembling a matrix or table, ideal for various data manipulation and analysis tasks. |
| Universal Functions | Universal functions (ufuncs) in NumPy are functions that operate element-wise on arrays, providing efficient and vectorized operations for a wide range of mathematical and logical operations. |
| Vector addition | Vector addition in Python involves adding corresponding elements of two or more vectors, producing a new vector with the sum of their components. |
| Visualizations | Visualizations in Python refer to the creation of graphical representations, such as charts, plots, and graphs, to illustrate and communicate data and trends visually. |

**Overview of HTTP**

When you, the **client**, use a web page your browser sends an **HTTP** request to the **server** where the page is hosted. The server tries to find the desired **resource** by default "index.html". If your request is successful, the server will send the object to the client in an **HTTP response**. This includes information like the type of the **resource**, the length of the **resource**, and other information.

The figure below represents the process. The circle on the left represents the client, the circle on the right represents the Web server. The table under the Web server represents a list of resources stored in the web server. In this case an HTML file, png image, and txt file .

The **HTTP** protocol allows you to send and receive information through the web including webpages, images, and other web resources. In this lab, we will provide an overview of the Requests library for interacting with the HTTP protocol.



**Uniform Resource Locator:URL**

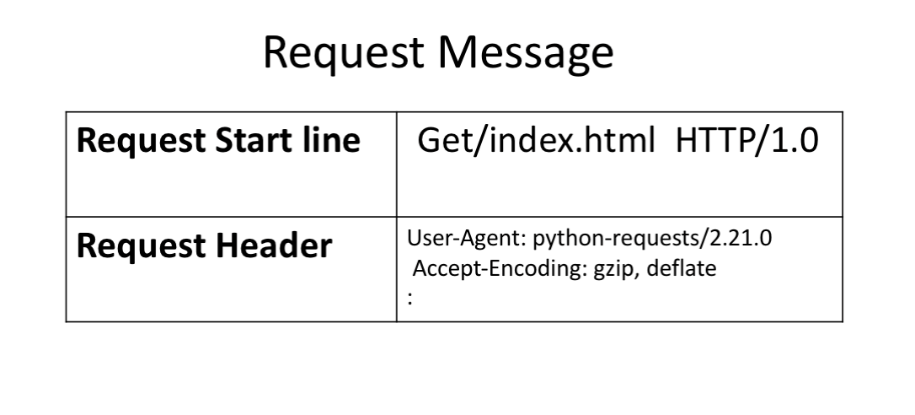
Uniform resource locator (URL) is the most popular way to find resources on the web. We can break the URL into three parts.

* **Scheme**:- This is this protocol, for this lab it will always be http://
* **Internet address or Base URL**:- This will be used to find the location here are some examples: www.ibm.com and  www.gitlab.com
* **Route**:- Location on the web server for example: /images/IDSNlogo.png

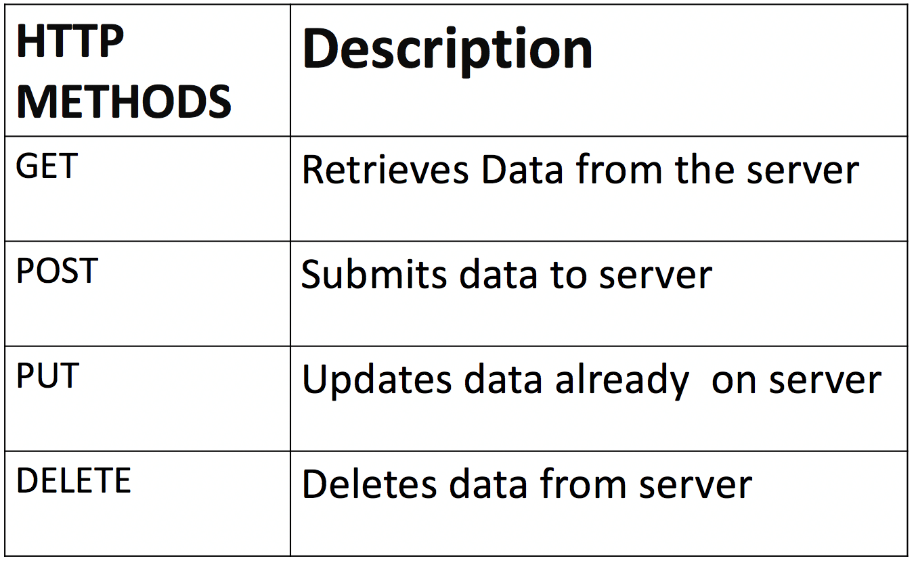
You may also hear the term Uniform Resource Identifier (URI), URL are actually a subset of URIs. Another popular term is endpoint, this is the URL of an operation provided by a Web server.

**Request**

The process can be broken into the **Request** and **Response**process. The request using the get method is partially illustrated below. In the start line we have the GET method, this is an HTTP method. Also the location of the resource /index.html and the HTTP version. The Request header passes additional information with an HTTP request:

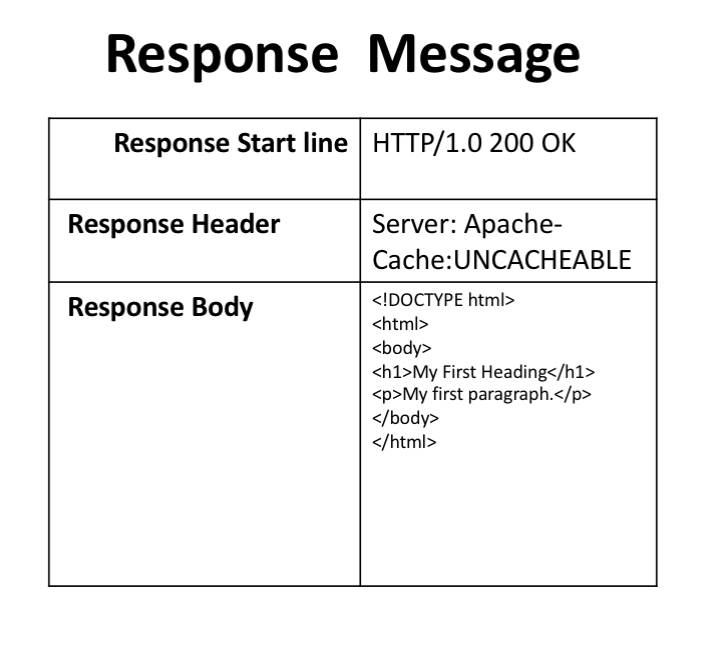


When an HTTP request is made, an HTTP method is sent, this tells the server what action to perform. A list of several HTTP methods is shown below. We will go over more examples later.

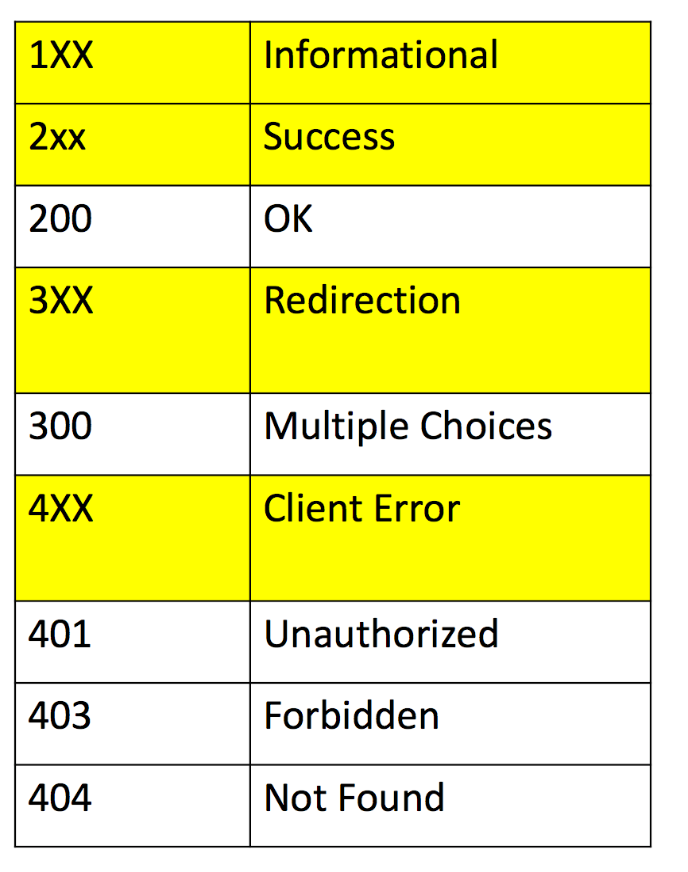


**Response**

The figure below represents the response; the response start line contains the version number HTTP/1.0, a status code (200) meaning success, followed by a descriptive phrase (OK). The response header contains useful information. Finally, we have the response body containing the requested file, an  HTML  document. It should be noted that some requests have headers.



Some status code examples are shown in the table below, the prefix indicates the class. These are shown in yellow, with actual status codes shown in white. Check out the following [link](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkPY0101ENSkillsNetwork19487395-2021-01-01)for more descriptions.



The purpose of this notebook is to provide more examples on how to use simple APIs. As you have already learned from previous videos and notebooks, API stands for Application Programming Interface and is a software intermediary that allows two applications to talk to each other.

The advantages of using APIs:

* **Automation**. Less human effort is required and workflows can be easily updated to become faster and more  
  productive.
* **Efficiency**. It allows to use the capabilities of one of the already developed APIs than to try to independently implement some functionality from scratch.

The disadvantage of using APIs:

* **Security**. If the API is poorly integrated, it means it will be vulnerable to attacks, resulting in data breeches or losses having financial or reputation implications.

One of the applications we will use in this notebook is Random User Generator. RandomUser is an open-source, free API providing developers with randomly generated users to be used as placeholders for testing purposes. This makes the tool similar to Lorem Ipsum, but is a placeholder for people instead of text. The API can return multiple results, as well as specify generated user details such as gender, email, image, username, address, title, first and last name, and more. More information on [RandomUser](https://randomuser.me/documentation" \l "intro" \t "_blank) can be found here.

Another example of simple API we will use in this notebook is Fruityvice application. The Fruityvice API web service which provides data for all kinds of fruit! You can use Fruityvice to find out interesting information about fruit and educate yourself. The web service is completely free to use and contribute to.

Bellow are Get Methods parameters that we can generate. For more information on the parameters, please visit this [documentation](https://randomuser.me/documentation) page.

**Get Methods**

* get\_cell()
* get\_city()
* get\_dob()
* get\_email()
* get\_first\_name()
* get\_full\_name()
* get\_gender()
* get\_id()
* get\_id\_number()
* get\_id\_type()
* get\_info()
* get\_last\_name()
* get\_login\_md5()
* get\_login\_salt()
* get\_login\_sha1()
* get\_login\_sha256()
* get\_nat()
* get\_password()
* get\_phone()
* get\_picture()
* get\_postcode()
* get\_registered()
* get\_state()
* get\_street()
* get\_username()
* get\_zipcode()