Contents

[PYTHON 3](#_Toc185683362)

[PYPI (PYTHON PACKAGE INDEX): 3](#_Toc185683363)

[ANACONDA 3](#_Toc185683364)

[RUNNING PYTHON FILE FROM COMMAND LINE (REPL – READ EVALUATE PRINT LOOP) 3](#_Toc185683365)

[PYTHON DATA TYPES 4](#_Toc185683366)

[OPERATORS 4](#_Toc185683367)

[TYPE CONVERSION 5](#_Toc185683368)

[TYPE CHECKING 5](#_Toc185683369)

[NUMBER MANIPULATION 6](#_Toc185683370)

[DATATYPES 6](#_Toc185683371)

[STRINGS 6](#_Toc185683372)

[ITERABLES 8](#_Toc185683373)

[LISTS 8](#_Toc185683374)

[DICTIONARIES 9](#_Toc185683375)

[TUPLES 10](#_Toc185683376)

[SETS 11](#_Toc185683377)

[ENUMERATE 11](#_Toc185683378)

[BOOLEANS 12](#_Toc185683379)

[VARIABLE SCOPE 12](#_Toc185683380)

[COMMENTS AND DOCSTRING 13](#_Toc185683381)

[COMMENTS 13](#_Toc185683382)

[DOCSTRINGS: 13](#_Toc185683383)

[STATEMENTS 13](#_Toc185683384)

[IF-ELSE 13](#_Toc185683385)

[NESTED IF-ELSE 14](#_Toc185683386)

[ELIF 14](#_Toc185683387)

[LOOPS 14](#_Toc185683388)

[FOR LOOP 14](#_Toc185683389)

[WHILE LOOP 15](#_Toc185683390)

[EXAMPLES 15](#_Toc185683391)

[I/O FILES 15](#_Toc185683392)

[WITH STATEMENT 16](#_Toc185683393)

[CURSORS IN FILE READING 17](#_Toc185683394)

[SOME USEFUL OPERATORS 17](#_Toc185683395)

[RANGE 17](#_Toc185683396)

[FUNCTIONS 18](#_Toc185683397)

[\*args and \*\*kwargs 19](#_Toc185683398)

[INTERACTIVITY – USER INPUTS 19](#_Toc185683399)

[LAMBDA EXPRESSIONS, MAP AND FILTERS 19](#_Toc185683400)

[MAP 20](#_Toc185683401)

[FILTERS 20](#_Toc185683402)

[NESTED STATEMENT AND SCOPE 20](#_Toc185683403)

[OOPS IN PYTHON 21](#_Toc185683404)

[CLASSES AND OBJECTS 21](#_Toc185683405)

[CLASS OBJECT ATTRIBUTES 21](#_Toc185683406)

[INHERITANCE 21](#_Toc185683407)

[POLYMORPHISM 22](#_Toc185683408)

[MODULES AND PACKAGES 22](#_Toc185683409)

[PIP INSTALL AND PyPi 22](#_Toc185683410)

[WRITING OWN PACKAGES AND MODULES 23](#_Toc185683411)

[\_\_name\_\_ and \_\_main\_\_ 24](#_Toc185683412)

[ERROR HANDLING 26](#_Toc185683413)

[MECHANISMS FOR ERROR HANDLING 26](#_Toc185683414)

[DECORATORS 26](#_Toc185683415)

[ASSIGNING FUNCTION TO VARIABLE 27](#_Toc185683416)

[ADVANCED MODULES 29](#_Toc185683417)

[COLLECTIONS MODULE 29](#_Toc185683418)

[WEB SCRAPING 32](#_Toc185683419)

[WEB SCRAPING PROCESS 32](#_Toc185683420)

[WORKING WITH PDFs AND SPREADSHEETS 33](#_Toc185683421)

# PYTHON

|  |
| --- |
| * Python distribution refers to a bundle or package that programming language along with additional tools, libraries, and packages that are commonly used includes the core Python programming language interpreter, along with in Python development. * It provides an easy way to install and manage Python and its related components, making it convenient for users to additional libraries, tools, and resources that are commonly used in Python development. It provides a convenient and pre-packaged environment for working with Python. |

## PYPI (PYTHON PACKAGE INDEX):

1. **Official package repository**
   1. PyPI is the official repository for Python packages and libraries.
   2. It hosts a vast collection of packages contributed by the Python community.
2. **General-purpose packages**:
   1. PyPI is suitable for a wide range of Python development needs, including web development, data analysis, machine learning, scientific computing, and more.
3. **`pip` package manager**:
   1. PyPI packages are typically installed using `pip`, the default package manager for Python.
   2. `pip` allows you to easily install, upgrade, and manage packages from PyPI and other package indexes.
4. **Community-driven**:
   1. PyPI has a large and active community of developers who contribute, maintain, and update packages.
   2. It is widely used and supported by the Python community.

ANACONDA:

1. **Distribution for data science**:
   1. Anaconda is a Python distribution **specifically focused on data science, scientific computing, and machine learning**. It includes a curated set of packages optimized for these domains.
2. **Conda package manager**:
   1. Anaconda comes with its own package manager called `conda`, which can install packages from Anaconda Cloud (Anaconda's package repository) as well as PyPI. `conda` provides additional features like environment management and handling binary dependencies.
3. **Pre-installed packages**:
   1. Anaconda comes with a comprehensive set of pre-installed packages commonly used in data science, such as NumPy, Pandas, Matplotlib, scikit-learn, and TensorFlow. This makes it convenient for data scientists to get started quickly.
4. **Environment management**:
   1. `conda` allows us to create and manage isolated environments with specific versions of Python and packages. This helps in managing different project requirements and avoiding conflicts between dependencies.

## RUNNING PYTHON FILE FROM COMMAND LINE (REPL – READ EVALUATE PRINT LOOP)

* Enter the “**python**” command to in terminal to start the Python **REPL**

|  |  |
| --- | --- |
|  | A white background with black text  Description automatically generated |

|  |  |
| --- | --- |
| RUNNING A PYTHON FILE | python <file\_name> |

## PYTHON DATA TYPES

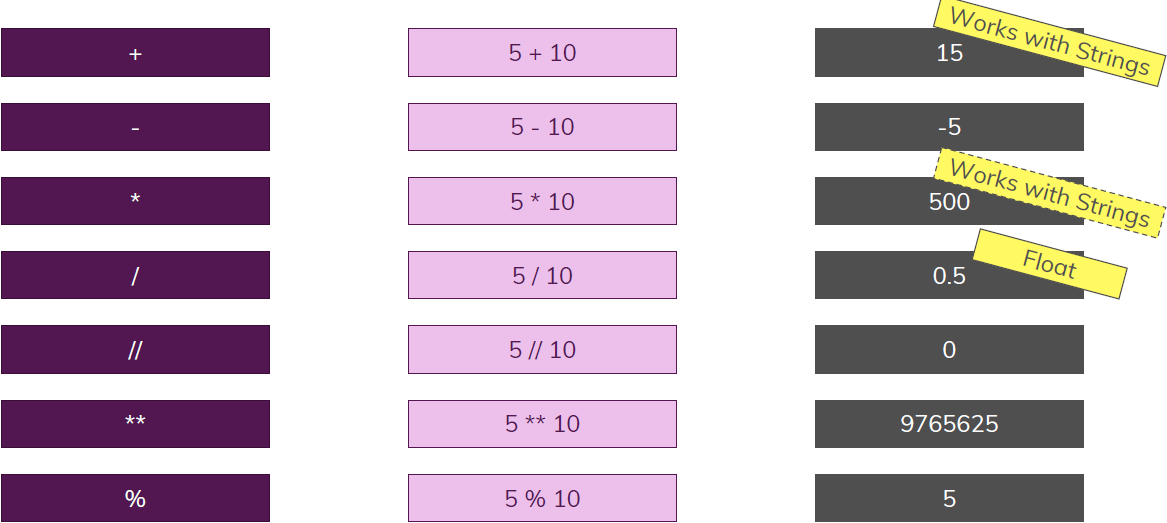
|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Example(s)** |
| **int** | Integer numbers without decimal points | x = 5 |
| **float** | Floating-point numbers with decimal points | y = 3.14 |
| **str** | Sequence of characters enclosed in single or double quotes | name = "John" |
| **bool** | Represents the truth values True or False | is\_valid = True |
| **list** | Ordered collection of items enclosed in square brackets | numbers = [1, 2, 3, 4] |
| **tuple** | Immutable ordered collection of items enclosed in parentheses | coordinates = (10, 20) |
| **dictionary** | Unordered collection of key-value pairs enclosed in curly braces | person = {'name': 'John', 'age': 25, 'city': 'NYC'} |
| **set** | Unordered collection of unique elements enclosed in curly braces | fruits = {'apple', 'banana', 'orange'} |

**NOTE**

* In Python, **we can represent large numbers using underscores as separators for improved readability**.
* This feature was introduced in Python 3.6.

|  |  |
| --- | --- |
| large\_number = 1\_000\_000\_000\_000\_000\_000  print(large\_number) | OUTPUT  1000000000000000000 |

## OPERATORS



**NOTE**

|  |  |
| --- | --- |
| * The double forward slash operator // is used for floor division. * Floor division performs division between two numbers and rounds down the result to the nearest integer. | a = 10  b = 3    result = a // b  print(result) 🡪 3 |
| * We can multiply a string by an integer to repeat the string multiple times. * The \* operator is used for string multiplication. | string = "Hello "  multiplier = 3    result = string \* multiplier  print(result)  **OUTPUT : Hello Hello Hello** |

## TYPE CONVERSION

|  |  |
| --- | --- |
| Python is a dynamically Typed language | In Python, dynamic typing allows variables to hold values of any data type and allows the type of a variable to be changed during execution |
| Example | x = 5  print(x) # Output: 5  print(type(x)) # Output: <class 'int'>  x = "Hello"  print(x) # Output: Hello  print(type(x)) # Output: <class 'str'> |

* In Python, type conversion refers to the process of converting one data type to another. Python provides several built-in functions for type conversion:

|  |  |
| --- | --- |
| **int(): Converts a value to an integer data type.**  Example:  num = int("10") # converts the string "10" to an integer  print(num) # output: 10 | **float(): Converts a value to a floating-point data type**.  Example:  num = float("3.14") # converts the string "3.14" to a float  print(num) # output: 3.14 |
| **str(): Converts a value to a string data type.**  Example:  age = 25  age\_str = str(age) # converts the integer 25 to a string  print(age\_str) # output: "25" | **list(): Converts a value to a list data type.**  Example:  numbers = "1 2 3 4 5"  numbers\_list = list(numbers)  # converts the string "1 2 3 4 5" to a list  print(numbers\_list)  # output: ['1', ' ', '2', ' ', '3', ' ', '4', ' ', '5'] |
| **tuple(): Converts a value to a tuple data type.**  Example:  numbers = "1 2 3 4 5"  # converts the string "1 2 3 4 5" to a tuple  numbers\_tuple = tuple(numbers)  print(numbers\_tuple)  # output: ('1', ' ', '2', ' ', '3', ' ', '4', ' ', '5') | **bool(): Converts a value to a boolean data type.**  Example:  value = 0  bool\_value = bool(value)  # converts the integer 0 to False  print(bool\_value)  # output: False |

## TYPE CHECKING

1. **Using the type() function: The type() function returns the data type of an object.**

|  |  |
| --- | --- |
| number = 10  print(type(number)) # output: <class 'int'> | name = "John"  print(type(name)) # output: <class 'str'> |
| is\_valid = True  print(type(is\_valid)) # output: <class 'bool'> |  |

1. **Using the isinstance() function: The isinstance() function checks if an object belongs to a specific class or data type. It returns True if the object is an instance of the specified class or data type, otherwise it returns False.**

|  |  |
| --- | --- |
| number = 10  print(isinstance(number, int)) # output: True | name = "John"  print(isinstance(name, str)) # output: True |
| is\_valid = True  print(isinstance(is\_valid, bool)) # output: True |  |

1. **Using the type annotations:** 
   1. **Python 3.5 and above support type annotations, which allow us to specify the expected data type of variables and function arguments.**
   2. **Type annotations are not enforced at runtime, but they can be used by static type checkers like Mypy to analyze the code for potential type errors.**

|  |
| --- |
| Example:  def add\_numbers(a: int, b: int) -> int:  return a + b  result = add\_numbers(5, 10)  print(result) # output: 15 |

Type checking is not mandatory in Python, as it is a dynamically typed language. However, it can help catch potential errors and make your code more robust and self-explanatory.

## NUMBER MANIPULATION

|  |  |
| --- | --- |
| **int(8/3)** | OUTPUT: 2  Trims the decimal points |
| **round(8/3,2)** | OUTPUT: 2.67  Round it to 2 decimal places |
| **8//3** | It is same as int(8/3) |
| **Assignment Operator** | score=0  score +=1 🡪 1  score -=1 🡪 0 |

## DATATYPES

### STRINGS

* In Python, a string is a sequence of characters enclosed in either single quotes (' ') or double quotes (" "). Strings are one of the basic data types in Python and are used to represent text or a sequence of characters.

#### STRINGS ARE IMMUTABLE

Strings are immutable. This means that once a string is created, we cannot change its individual characters. However, we can create a new string by performing operations on the original string.

|  |
| --- |
| my\_string = "Hello"  my\_string[0] = "J" # This will raise a TypeError: 'str' object does not support item assignment |
| **SOLUTION**  my\_string = "Hello"  new\_string = "J" + my\_string[1:]  print(new\_string) # Output: Jello |

#### STRINGS OPERATIONS

|  |  |  |
| --- | --- | --- |
| **CREATING A STRING**  my\_string = "Hello, World!"  print(my\_string) # Output: Hello, World! | **ACCESSING CHARACTERS IN A STRING**  my\_string = "Hello, World!"  print(my\_string[0]) # Output: H  print(my\_string[7]) # Output: W | **STRING LENGTH**  my\_string = "Hello, World!"  print(len(my\_string)) # Output: 13 |
| **CONCATENATING STRINGS**  string1 = "Hello"  string2 = "World"  concatenated\_string = string1 + " " + string2  print(concatenated\_string) # Output: Hello World | **STRING SLICING**  my\_string = "Hello, World!"  print(my\_string[7:12]) # Output: World  print(my\_string[:5]) # Output: Hello  print(my\_string[7:]) # Output: World! | **STRING METHODS**  my\_string = "Hello, World!"  print(my\_string.upper()) # Output: HELLO, WORLD!  print(my\_string.lower()) # Output: hello, world!  print(my\_string.split(",")) # Output: ['Hello', ' World!']  print(my\_string.replace("Hello", "Hi")) # Output: Hi, World! |
| **LONG STRING**   * triple quotes (""" or ''') are used to define multi-line strings or string literals. * Triple quotes allow us to include line breaks and preserve the formatting of the text within the string. | | **my\_string = """This is a**  **multi-line**  **string."""**  **print(my\_string)**  **OUTPUT**  **This is a**  **multi-line**  **string.** |
|  | |  |

#### STRING FORMATTING

##### FORMAT METHOD

* The format() method is used to format strings by replacing placeholders with corresponding values.

|  |  |
| --- | --- |
| **BASIC SYNTAX** | formatted\_string = "Hello, {}!".format(value) |
| * The curly braces {} serve as placeholders in the string. * The format() method replaces these placeholders with the value specified inside the format() method.   name = "Alice"  age = 25  message = "Hello, my name is {} and I am {} years old.".**format**(name, age)  print(message) # Output: Hello, my name is Alice and I am 25 years old. | |
| **EXAMPLE - 1**  name = "John"  age = 25  **message = f"My name is {name} and I am {age} years old."**  **print(message) # output: My name is John and I am 25 years old.** | |
| **EXAMPLE -2**  name1='Alex' name2='Pam' name2='Alok' print("Friends name {1},{2} and {0}".format(name1,name1,name2))  **# output: Friends name Alex,Alok and Alex.** | |
| **EXAMPLE - 3**  name1='Alex' name2='Pam' name2='Alok' print("Friends name {secondName},{thirdName} and {firstName}".format(firstName=name1,secondName=name1,thirdName=name2))  **# output: Friends name Alex,Alok and Alex** | |
| pi = 3.14159  formatted\_pi = "The value of pi is approximately **{:.2f}".format(pi)**  print(formatted\_pi)  # Output: The value of pi is approximately 3.14 | |

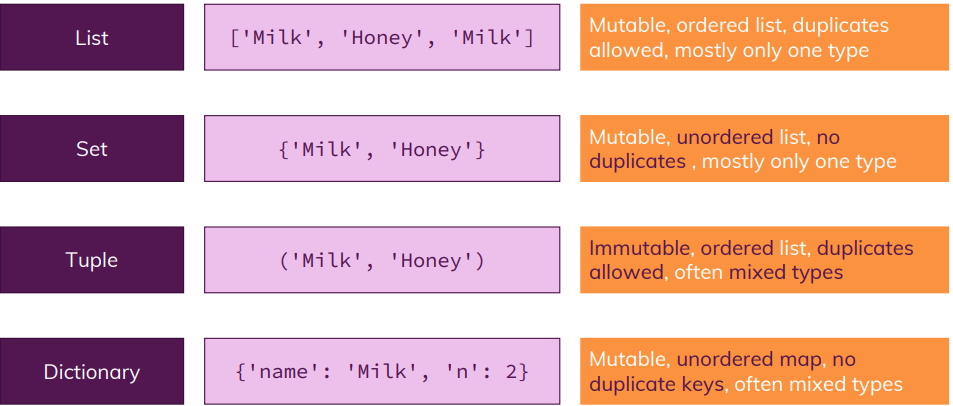
##### FSTRING

* f-strings (formatted string literals) provide a concise and convenient way to embed expressions inside string literals.
* They allow us to include variables, expressions, and even function calls within curly braces `{}` directly in the string.
* F-strings also support simple expressions and function calls:

|  |
| --- |
| x = 5  y = 10  result = f"The sum of {x} and {y} is {x + y}."  print(result) # output: The sum of 5 and 10 is 15.  greeting = f"Hello, {get\_name()}!"  print(greeting) # output: Hello, John! |

In the above examples, the expressions `{x + y}` and `{get\_name()}` are evaluated and the results are inserted into the strings.

## ITERABLES



### LISTS

* List is a data structure that holds an ordered collection of elements.
* Lists are mutable, which means we can change their content.

|  |  |
| --- | --- |
| **CREATING A LIST**  fruits = ["apple", "banana", "orange", "grape"] | **ACCESSING ELEMENTS**  print(fruits[0]) # Output: "apple"  print(fruits[2]) # Output: "orange" |
| **MODIFYING ELEMENTS**  fruits[1] = "mango"  print(fruits) # Output: ["apple", "mango", "orange", "grape"] | **ADDING ELEMENTS**  fruits.append("kiwi")  print(fruits) # Output: ["apple", "mango", "orange", "grape", "kiwi"] |
| **REMOVING ELEMENTS**  fruits.remove("orange")  print(fruits) # Output: ["apple", "mango", "grape", "kiwi"]  **REMOVE THE LAST ELEMENT FROM THE LIST**  print(fruits.pop()) *#grape*  **REMOVES THE SPECIFIC INDEX ELEMENT FROM THE LIST**  fruits = ["apple", "banana", "orange", "grape"]  poped\_element = fruits.pop(2) print(poped\_element) *#orange* | **CHECKING IF AN ELEMENT EXISTS**  print("banana" in fruits) # Output: True |
| **# Length of the list**  print(len(fruits)) # Output: 4 |
| **ITERATING OVER A LIST**  for fruit in fruits:  print(fruit) |
| **LAST VALUE IN LIST**  blocklist = [1,2,3]  blocklist[1] 🡪 3 |
| **SLICING**  fruits=['apple', 'mango', 'orange']; print(fruits[1:]) *#['mango', 'orange']* print(fruits[1:2]) *# ['mango']* print(fruits[:2]) *#['apple', 'mango']* | **CONCATENATION**  fruits=['apple', 'mango', 'orange']; dryFruits =['almond','peanuts'] print(fruits+dryFruits) *#['apple', 'mango', 'orange', 'almond', 'peanuts']* |

#### LIST COMPREHENSION

* List comprehension is a concise way to create lists in Python. It allows us to generate a new list by applying an expression to each element of an existing iterable (such as a list, tuple, or string), along with optional filtering conditions. List comprehensions provide a more readable and compact alternative to using traditional loops.

|  |  |
| --- | --- |
| SYNTAX | new\_list = [expression for item in iterable if condition]  **Example :**  new\_list = [number for number in range(1,10) if number%2 ==0] print(new\_list)   * **expression**`: The expression that is applied to each item in the iterable to generate the new list. . * `**item**`: The variable that represents each item in the iterable. * `**iterable**`: The existing iterable from which the items are taken. * `**condition**` (optional): A condition that filters the items. Only the items for which the condition evaluates to True are included in the new list. |

EXAMPLES

|  |  |
| --- | --- |
| **SQUARING NUMBERS FROM 1 TO 5** | numbers = [1, 2, 3, 4, 5]  squared\_numbers = [num \*\* 2 for num in numbers]  print(squared\_numbers) # Output: [1, 4, 9, 16, 25] |
| **FILTERING EVEN NUMBERS FROM A LIST** | numbers = [1, 2, 3, 4, 5]  even\_numbers = [num for num in numbers if num % 2 == 0]  print(even\_numbers) # Output: [2, 4 |
| **EXTRACTING VOWELS FROM A STRING** | text = "Hello, World!"  vowels = [char for char in text if char.lower() in 'aeiou']  print(vowels) # Output: ['e', 'o', 'o'] |
| **Use List Comprehension to create a list of the first letters of every word in the string below:** | st = 'Create a list of the first letters of every word in this string' words = [ letter for letter in st.split()] firstLetterWord = [chars[0] for chars in words] print(firstLetterWord) |
| **Use a List Comprehension to create a list of all numbers between 1 and 50 that are divisible by 3.** | numberDivisibleBy3= [num for num in range(1,51) if num %3 ==0 ] print(numberDivisibleBy3) |

### DICTIONARIES

* Dictionary is a collection of key-value pairs that are unordered and changeable.

|  |
| --- |
| **# Creating a dictionary**  person = {  "name": "John",  "age": 30,  "city": "New York"  }    **# Accessing values**  print(person["name"]) # Output: "John"  print(person["age"]) # Output: 30    **# Modifying values**  person["age"] = 32  print(person) # Output: {'name': 'John', 'age': 32, 'city': 'New York'}    **# Adding new key-value pairs**  person["occupation"] = "Engineer"  print(person) # Output: {'name': 'John', 'age': 32, 'city': 'New York', 'occupation': 'Engineer'}    **# Removing key-value pairs**  del person["city"]  print(person) # Output: {'name': 'John', 'age': 32, 'occupation': 'Engineer'}    **# Checking if a key exists**  print("age" in person) # Output: True    **# Length of the dictionary**  print(len(person)) # Output: 3    **# Iterating over key-value pairs**  for key, value in person.items():  print(key, value)  **#print all keys/ values / Key and value**  print("values",person.values()) 🡪 #output dict\_values(['Alex', 27, 'New York']) print("keys",person.keys()) 🡪 # output : dict\_keys(['name', 'age', 'city'])  print(person.items()) 🡪 # output - dict\_items([('name', 'Alex'), ('age', 27), ('city', 'New York')]) |

### TUPLES

* A tuple is an immutable collection of **ordered** elements.
* It is like a list, but unlike lists, tuples cannot be modified once created.
* Tuples are created using parentheses () or the tuple() constructor.

|  |
| --- |
| **# Creating a tuple**  fruits = ("apple", "banana", "orange", "grape")    **# Accessing elements**  print(fruits[0]) # Output: "apple"  print(fruits[2]) # Output: "orange"    **# Cannot modify elements (immutable)**  fruits[1] = "mango" # Raises TypeError    **# Length of the tuple**  print(len(fruits)) # Output: 4    **# Iterating over a tuple**  for fruit in fruits:  print(fruit)    **# Tuple packing and unpacking**  name = "John"  age = 30  person = (name, age)  print(person) # Output: ("John", 30)    name, age = person  print(name) # Output: "John"  print(age) # Output: 30  ## Number of times an element in a tuple  letters = ('a','a','c','d') print(letters.count('a')) #output 2  #index of an element in tuple  letters = ('a','b','c','d') print(letters.index('c')) #output 2 |

### SETS

* **Set is an unordered collection of unique elements**.
* Sets are created using the set() function or by enclosing a comma-separated sequence of elements within curly braces {}.

|  |  |
| --- | --- |
| **CREATING A SET** | my\_set = set() # Empty set  my\_set = {1, 2, 3} # Set with elements 1, 2, and 3 |
| **ADDING ELEMENTS TO A SET:** | my\_set.add(4) # Adds element 4 to the set  my\_set.update([5, 6, 7]) # Adds multiple elements to the set |
| **REMOVING ELEMENTS FROM A SET:** | my\_set.remove(3) # Removes element 3 from the set  my\_set.discard(4) # Removes element 4 if it exists, otherwise does nothing  my\_set.pop() # Removes and returns an arbitrary element from the set |
| **SET OPERATIONS:** | set1 = {1, 2, 3}  set2 = {3, 4, 5}  **RETURNS A SET CONTAINING ALL ELEMENTS FROM BOTH SETS**  union\_set = set1.union(set2)  **# RETURNS A SET CONTAINING COMMON ELEMENTS FROM BOTH SETS**  intersection\_set = set1.intersection(set2)  **# RETURNS A SET CONTAINING ELEMENTS PRESENT IN SET1 BUT NOT IN SET2**  difference\_set = set1.difference(set2)  **# RETURNS A SET CONTAINING ELEMENTS PRESENT IN EITHER SET1 OR SET2, BUT NOT IN BOTH**  symmetric\_difference\_set = set1.symmetric\_difference(set2) |
| **OTHER USEFUL METHODS:** | len(my\_set) # Returns the number of elements in the set  element in my\_set # Checks if an element is present in the set  my\_set.clear() # Removes all elements from the set . |
| **SET FOR STRINGS** | * The set() function can be used to create a set from an iterable object, such as a string. * When we pass a string to the set() function, it treats the string as an iterable sequence of characters and creates a set containing all the unique characters from the string.   my\_set = set('Parallel')  print(my\_set) #output {'P', 'a', 'r', 'l', 'e'}   * Note that sets are unordered collections, so the order of the elements in the set may not match the original order of the characters in the string. * Additionally, since sets only contain unique elements, any duplicate characters in the string will be removed in the resulting set. |

### ENUMERATE

|  |  |
| --- | --- |
| * `enumerate()` function is a built-in function that allows us to loop over an iterable (e.g., list, tuple, string) while also keeping track of the index of each item. * It returns an enumerate object, which can be converted to a list of tuples or used directly in a loop. | |
| **EXAMPLE**    fruits = ['apple', 'banana', 'orange']  # Using enumerate in a loop  for **index, fruit** in enumerate(fruits):  print(index, fruit)  **Output:**  ```  0 apple  1 banana  2 orange | * enumerate(fruits)` returns an enumerate object that yields tuples containing the index and value of each item in the `fruits` list. * The `for` loop iterates over the enumerate object. * In each iteration, the `index` variable receives the index of the current item, and the `fruit` variable receives the value of the current item. * The `print()` statement displays the index and fruit values for each iteration. * Note that the index starts from 0 by default, but you can specify a different starting index by passing a second argument to `enumerate()`. For example, `enumerate(fruits, start=1)` will start the index from 1 instead of 0. |

### BOOLEANS

* Boolean data type represents the truth values `True` and `False`.
* The boolean type has two possible values: `True` and `False`. These values are case-sensitive and must be written in title case.

#### LOGICAL OPERATORS – CHAINING COMPARISON OPERATOR

* Booleans can be combined using logical operators to create more complex conditions. The logical operators include:

|  |  |
| --- | --- |
| * *`and`: Returns `True` if both operands are `True`.* * *`or`: Returns `True` if at least one of the operands is `True`.* * *`not`: Returns the opposite boolean value of the operand.* | is\_raining = True  is\_sunny = False  print(is\_raining and is\_sunny) # False  print(is\_raining or is\_sunny) # True  print(not is\_raining) # False |

BOOLEAN FUNCTIONS

* Python provides built-in functions that can be used to work with booleans, such as:

|  |  |
| --- | --- |
| * *`****bool****()`: Converts a value to a boolean. Returns `False` for empty containers, `0`, and `None`; otherwise, returns `True`.* * *`****all****()`: Returns `True` if all elements in an iterable are `True`.* * *`****any****()`: Returns `True` if at least one element in an iterable is `True`.* | x = 10  y = 0  z = []  print(bool(x)) # True  print(bool(y)) # False  print(bool(z)) # False  numbers = [1, 2, 3, 4, 5]  print(all(numbers)) # True  print(any(numbers)) # True |

## VARIABLE SCOPE

* The `**global**` keyword is used to indicate that a variable inside a function should be treated as a global variable, rather than a local variable.
* When we use the `global` keyword before a variable assignment inside a function, it tells Python that we want to modify the value of a global variable, instead of creating a new local variable with the same name.

|  |  |
| --- | --- |
| global\_var = 10  def my\_function():  global global\_var  global\_var = 20  print(global\_var)  my\_function() # Output: 20  print(global\_var) # Output: 20 | * The `global` keyword is used inside the `my\_function` to indicate that `global\_var` refers to the global variable, not a local variable. * Thus, the assignment `global\_var = 20` modifies the value of the global variable, and both the function and the outside code see the updated value. |

## COMMENTS AND DOCSTRING

* Comments and docstrings are used to provide explanations, document code, and make it more readable. While both serve a similar purpose, they have different formats and use cases.

COMMENTS

|  |  |
| --- | --- |
| Comments start with the `#` symbol and continue until the end of the line. | # This is a comment explaining the purpose of the following code  x = 10 # This is a comment explaining the value of x |

### DOCSTRINGS:

* Docstrings, short for "documentation strings," are used to provide documentation for functions, classes, or modules. They are enclosed in triple quotes (`"""` or `'''`) and are typically placed immediately after the function, class, or module definition.

|  |  |
| --- | --- |
| * Docstrings provide a standardized way to document code and convey information about the purpose, parameters, return values, and usage of functions, classes, or modules. They can be accessed using the `\_\_doc\_\_` attribute of an object. * To generate well-documented code, it's good practice to include docstrings for all public functions, classes, and modules. * Tools like Sphinx can automatically generate documentation based on docstrings. | def greet(name):  """  A function that takes a name as input and prints a greeting message.    Parameters:  name (str): The name of the person to greet.    Returns:  None  """  print(f"Hello, {name}!")    greet("Alice") |

## STATEMENTS

### IF-ELSE

|  |  |
| --- | --- |
| height = int(input("What's your Height? ")) if height > 120:  print("Can ride the Rollercoaster") else:  print("Can't ride the Rollercoaster") | * Indentation of if / else block is important. * “:” character after the if and else . |
| num = int(input("Enter a number: "))  if num % 2 == 0:  print("Even")  else:  print("Odd") | * Remainder Operator |

### NESTED IF-ELSE

|  |
| --- |
| height = int(input("What's your Height? "))  if height > 120:  print("Can ride the Rollercoaster")  age = int(input("Enter your age: "))  if age < 18:  print("Please pay $7")  else:  print("Please pay $12")  else:  print("Can't ride the Rollercoaster") |

### ELIF

|  |
| --- |
| height = int(input("What's your Height? "))  if height > 120:  print("Can ride the Rollercoaster")  age = int(input("Enter your age: "))  if age < 12:  print("Please pay $5")  elif age <= 18:  print("Please pay $7")  else:  print("Please pay $12")  else:  print("Can't ride the Rollercoaster") |

## LOOPS

### FOR LOOP

* for loop is used to iterate over a sequence or other iterable objects, such as lists, strings, or tuples.

|  |  |
| --- | --- |
| **ITERATING OVER A LIST** | fruits = ["apple", "banana", "cherry"]  for fruit in fruits:  print(fruit) |
| **ITERATING OVER A STRING** | message = "Hello, World!"  for char in message:  print(char) |
| **ITERATING OVER A RANGE OF NUMBERS:** | for i in range(1, 6):  print(i) |
| **SKIPPING ITERATIONS WITH CONTINUE:** | numbers = [1, 2, 3, 4, 5]  for num in numbers:  if num == 3:  **continue**  print(num) |
| **LIST COMPREHENSION** | nums = [1,2,3,4,5,6,7,8,9,10]  [num for num in nums if num %2 ==0]  **OUTPUT : [2, 4, 6, 8, 10]** |
| **TERMINATING THE LOOP WITH BREAK:** | numbers = [1, 2, 3, 4, 5]  for num in numbers:  if num == 3:  **break**  print(num) |
| **TUPLE UNPACKING** | mylist = [(1,2),(3,4),(5,6)] for value1, value2 in mylist:  print("value 1 ={} and value2= {}".format(value1, value2 ))  **OUTPUT** value 1 =1 and value2= 2  value 1 =3 and value2= 4  value 1 =5 and value2= 6 |

### WHILE LOOP

|  |  |
| --- | --- |
| **SIMPLE WHILE** | count = 0  while count < 5:  print(count)  count += 1 |
| **WHILE WITH BREAK** | while True:  num = int(input("Enter a number (0 to exit): "))  if num == 0:  break  print("You entered:", num) |
| **WHILE WITH CONTINUE** | count = 0  while count < 5:  count += 1  if count == 3:  continue  print(count) |

### EXAMPLES

|  |  |
| --- | --- |
| **Use for, .split(), and if to create a Statement that will print out words that start with 's':** | st = 'Print only the words that start with s in this sentence' for s in st.split() :  if(s.startswith("s")):  print(s) |
| **Go through the string below and if the length of a word is even print "even!"** | st = 'Print every word in this sentence that has an even number of letters' for s in st.split():  if(len(s) %2 == 0):  print(“even”) |
|  |  |

## I/O FILES

|  |  |
| --- | --- |
| **OPENING A FILE** | * To open a file, we can use the `open()` function. * It takes the file path and a mode as arguments. The mode can be **`'r'` for reading, `'w'` for writing, `'a'` for appending, or `'x'` for creating a new file**.   **file = open("example.txt", "r")** |
| **READING FROM A FILE** | * To read the contents of a file, we can use the `read()` or `readline()` methods of the file object. * The `read()` method reads the entire file, while the `readline()` method reads one line at a time   **file = open("example.txt", "r")**  **# Reading the entire file**  **content = file.read()**  **print(content)**  **# Reading one line at a time**  **line = file.readline()**  **print(line)**  **file.close()** |
| **READLINES** | * The `readlines()` method is another way to read a file. * It reads all the lines of a file and returns them as a list of strings. Each string represents a line from the file, including the newline character (`'\n'`) at the end of each line.   **with open("example.txt", "r") as file:**  **lines = file.readlines()**    **for line in lines:**  **print(line)**    *In this example, the `readlines()` method is used to read all the lines from the file "example.txt".* ***The lines are stored in the `lines` variable as a list of strings****. The `for` loop is then used to iterate over each line and print it.*    **Note**: that each line retrieved from `readlines()` includes the newline character (`'\n'`) at the end. If we want to remove the newline character, we can use the `strip()` method on each line:  **with open("example.txt", "r") as file:**  **lines = file.readlines()**    **for line in lines:**  **line = line.strip()**  **print(line)** |
| **WRITING TO A FILE** | * To write to a file, we can use the `write()` method of the file object. * It writes the given content to the file. If the file doesn't exist, it creates a new file. If the file already exists, it overwrites the existing content   **file = open("example.txt", "w")**  **file.write("Hello, World!")**  **file.close()** |
| **APPENDING TO A FILE** | * To append content to an existing file, you can open the file in append mode (`'a'`) and then use the `write()` method to write the content   **file = open("example.txt", "a")**  **file.write("This is additional content.")**  **file.close()** |
| **CLOSING A FILE** | * After performing I/O operations on a file, it's important to close the file to release system resources. * We can use the `close()` method of the file object to close the file   **file = open("example.txt", "r")**  **content = file.read()**  **print(content)**  **file.close()** |

### WITH STATEMENT

|  |  |
| --- | --- |
| * We can use the `with` statement when working with files, as it automatically takes care of closing the file. * In this example, the file is automatically closed when the `with` block is exited, even if an exception occurs. | with open("example.txt", "r") as file:  content = file.read()  print(content) |

### CURSORS IN FILE READING

* When reading a file in Python, we can use cursors to control the position within the file.
* The cursor, also known as the file pointer, keeps track of the current position in the file from which the next read operation will start.

#### CURSOR-RELATED METHODS FOR FILE READING

|  |  |
| --- | --- |
| **seek(offset[, whence])`: Sets the cursor's position in the file.**   * `**offset**`: The number of bytes to move. Positive offsets move forward, and negative offsets move backward. * `**whence**` (optional): Specifies the reference position for the offset. Default is 0 (beginning of the file).   + `0`: Beginning of the file   + `1`: Current position   + `2`: End of the file | **tell()`: Returns the current position of the cursor in the file.** |

**EXAMPLE**

|  |  |
| --- | --- |
| * In this example, the `seek()` method is used to move the cursor to different positions within the file. * The `tell()` method is used to retrieve the current position of the cursor. * The `readline()` method is used to read the first line, * `read()` method is used to read the entire file or a specific number of characters. * Finally, the cursor is moved to the end of the file, and the `read()` method is used again to read the last 5 characters. | **with open("example.txt", "r") as file:**  **# Read the first line**  **line1 = file.readline()**  **print(line1)**    **# Get the current position of the cursor**  **position = file.tell()**  **print("Current position:", position)**    **# Move the cursor to the beginning of the file**  **file.seek(0)**    **# Read the entire file starting from the beginning**  **content = file.read()**  **print(content)**    **# Move the cursor to the end of the file**  **file.seek(0, 2)**    **# Read the last 5 characters**  **last\_chars = file.read(5)**  **print("Last 5 characters:", last\_chars)** |

#### MODES

## SOME USEFUL OPERATORS

### RANGE

In Python, the `range()` function is used to generate a sequence of numbers. It returns an iterable object that represents a sequence of numbers within a specified range. The basic syntax of the `range()` function is as follows:

```python

range(start, stop, step)

```

where:

- `start` (optional): The starting value of the sequence. If not specified, the default value is 0.

- `stop`: The stopping value of the sequence (exclusive). The `range()` function will generate numbers up to, but not including, this value.

- `step` (optional): The increment between numbers in the sequence. If not specified, the default value is 1.

Here are some examples to illustrate the usage of the `range()` function:

1. Generating a sequence from 0 to 4 (exclusive):

```python

for i in range(5):

print(i)

```

Output:

```

0

1

2

3

4

```

2. Generating a sequence from 2 to 8 (exclusive) with a step of 2:

```python

for i in range(2, 9, 2):

print(i)

```

Output:

```

2

4

6

8

```

3. Creating a list of numbers within a range:

```python

my\_list = list(range(1, 6))

print(my\_list)

```

Output:

```

[1, 2, 3, 4, 5]

```

The `range()` function is commonly used in loops to iterate over a sequence of numbers. It provides a convenient way to generate and work with ranges of numbers in Python.

## FUNCTIONS

|  |  |
| --- | --- |
| **FUNCTION SYNTAX** | def greet():  print("Hello, world!")  # Call the function  greet() |
| **FUNCTION WITH PARAMETERS** | def add\_numbers(a, b):  return a + b    # Call the function and store the result in a variable  result = add\_numbers(3, 5)  print(result) # Output: 8 |
| **DEFAULT VALUES FOR FUNCTION PARAMETERS** | def greet(**name="World"**):  print("Hello, " + name + "!")    # Call the function without passing any arguments  greet() # Output: Hello, World!    # Call the function with an argument  greet("Alice") # Output: Hello, Alice! |

### \*args and \*\*kwargs

|  |  |
| --- | --- |
| * The \*args syntax is used when we want to pass a variable number of non-keyword arguments to a function. * The args parameter is treated as a tuple that holds the additional arguments passed to the function. Here's an example: | def sum\_numbers(\*args):  total = 0  for num in args:  total += num  return total    result = sum\_numbers(1, 2, 3, 4, 5)  print(result) # Output: 15 |
| * The \*\*kwargs syntax is used when we want to pass a variable number of keyword arguments to a function. * The kwargs parameter is treated as a dictionary that holds the additional arguments passed to the function. Here's an example: | def print\_person\_info(\*\*kwargs):  for key, value in kwargs.items():  print(key + ": " + value)    print\_person\_info(name="Alice", age="25", city="New York") |

## INTERACTIVITY – USER INPUTS

|  |  |
| --- | --- |
| * The input() function in Python is used to accept user input from the keyboard. * It reads a line of text entered by the user and returns it as a string. | def user\_choice():  is\_invalid\_choice = True  while is\_invalid\_choice:  choice = input("Enter your choice (1-10): ")  if choice.isdigit() and 1 <= int(choice) <= 10:  is\_invalid\_choice = False  return int(choice)  else:  print("Invalid choice. Please enter a number between 1 and 10.") print(user\_choice()) |

## LAMBDA EXPRESSIONS, MAP AND FILTERS

* Lambda expressions are anonymous functions that can be created on the fly without using the def keyword.
* They are typically used for simple functions that are not needed elsewhere.
* They are used in conjunction with other functions like “map()” and “filter()”

|  |  |
| --- | --- |
| SYNTAX | lambda arguments: expression |
| EXAMPLE | # Create a lambda function that adds two numbers  add\_numbers = lambda x, y: x + y  # Call the lambda function  result = add\_numbers(5, 3)  print(result) # Output: 8 |

### MAP

* The map() function applies a given function to each item of an iterable and returns a new iterator with the results.

|  |  |
| --- | --- |
| **EXAMPLE** | def square(num):  return num \*\*2 for sq in map(square,range(1,5)):  print(sq) |
|  | USING LAMBDA EXPRESSION  # Convert a list of numbers to their squares using map()  numbers = [1, 2, 3, 4, 5]  squares = list(map(lambda x: x \*\* 2, numbers))  print(squares) # Output: [1, 4, 9, 16, 25] |
| **REVERSE** | names =["Andy","Dick","reef"] nameFirstLetter =list(map(lambda name:name[::-1],names)) print(nameFirstLetter) |

### FILTERS

* The filter() function creates an iterator from an iterable, including only the items that satisfy a given condition (specified by a lambda function or another function).

|  |  |
| --- | --- |
| **EXAMPLE** | # Filter even numbers from a list using filter()  evens = list(filter(lambda even:even % 2 ==0, list(range(1,51)))) print(evens) |

## NESTED STATEMENT AND SCOPE

* The Python LEGB rule, also known as the "**scope resolution rule**," is used to determine the order in which Python searches for variables in different scopes.
* The acronym LEGB stands for **Local, Enclosing, Global, and Built-in**, representing the four different scopes in Python.
* **Local (L):**
  + This refers to the local scope, which is the innermost scope.
  + It includes variables defined within a function.
  + **Local variables can only be accessed within the function in which they are defined.**
* **Enclosing (E):**
  + This refers to the scope of an enclosing function.
  + It applies to nested functions, where an inner function can access variables from its containing outer function.
* **Global (G):** 
  + This refers to the global scope, which includes variables defined at the top level of a module or explicitly declared as global using the `global` keyword.
  + Global variables can be accessed from anywhere within the module.
* **Built-in (B):**
  + This is the outermost scope and includes Python's built-in functions and modules.
  + These are pre-defined names and can be accessed from any module or function.
* **The LEGB rule specifies the order in which Python searches for a variable. It starts with the local scope🡪, then moves to the enclosing scope 🡪 followed by the global scope 🡪 and finally the built-in scope.**
* **If a variable is not found in any of these scopes, a `NameError` is raised.**

## OOPS IN PYTHON

## CLASSES AND OBJECTS

* A class is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will have.

|  |  |
| --- | --- |
| class Car():  def \_\_init\_\_(self, color, make):  self.color = color;  self.make = make;  def startEngine(self):  print("Time to start the {} which is {} in color".format(self.make,self.color))  honda = Car('blue', "city"); toyota = Car(make='Corolla',color="white") honda.startEngine() toyota.startEngine() | * \_\_init\_\_() is a special method, also known as the **constructor**, that is automatically called when an object is created from a is always **self**, which refers to the instance of the class being created. Additional parameters can be included to accept values that will be used to initialize the object's attributes. * The \_\_init\_\_() method does not explicitly return a value. Its purpose is to initialize the object class. It is used to initialize the attributes of the object. |

### CLASS OBJECT ATTRIBUTES

EXAMPPLE

|  |  |
| --- | --- |
| * Class object attributes are attributes that are defined at the class level and are shared by all instances of the class. * These attributes are associated with the class itself, rather than with any specific instance of the class. * Class object attributes are accessed using the class name, rather than an instance of the class | class Circle:  # Class object attribute  pi = 3.14159    def \_\_init\_\_(self, radius):  self.radius = radius    def calculate\_area(self):  return Circle.pi \* (self.radius \*\* 2)    # Accessing the class object attribute  print(Circle.pi) # Output: 3.14159    **# Creating instances of the Circle class**  circle1 = Circle(5)  circle2 = Circle(10)    **# Accessing the instance attribute**  print(circle1.radius) # Output: 5  print(circle2.radius) # Output: 10    **# Accessing the class object attribute through an instance**  print(circle1.pi) # Output: 3.14159  print(circle2.pi) # Output: 3.14159    # Calculating the area using the instance method  print(circle1.calculate\_area()) # Output: 78.53975  print(circle2.calculate\_area()) # Output: 314.159 |

### INHERITANCE

|  |  |
| --- | --- |
| class Car():  def \_\_init\_\_(self, color, make):  self.color = color;  self.make = make;  def startEngine(self):  print("Time to start the {} which is {} in color".format(self.make,self.color))  class EVCar(Car):  def \_\_init\_\_(self, color, make, batteryCapacity):  super().\_\_init\_\_(color,make)  self.batteryCapacity = batteryCapacity;  honda = Car('blue', "city"); toyota = Car(make='Corolla',color="white") tesla= EVCar("grey","Model S", 75)  honda.startEngine() toyota.startEngine() tesla.startEngine() | EVCar class:   * The EVCar class is the subclass or derived class of Car. * **It has its own \_\_init\_\_() method that uses the super() function to call the \_\_init\_\_() method of the superclass (Car) and pass the color and make arguments.** * By inheriting from the Car class, the EVCar class automatically gains access to the \_\_init\_\_() and startEngine() methods defined in Car. |

### POLYMORPHISM

|  |
| --- |
| class Animal():  def \_\_init\_\_(self, name):  self.name = name   def speak(self):  pass  class Dog(Animal):  def \_\_init\_\_(self, name):  super().\_\_init\_\_(name)   def speak(self):  print("{} speak WOFF!!".format(self.name))  class Cat(Animal):  def \_\_init\_\_(self, name):  super().\_\_init\_\_(name)   def speak(self):  print("{} speak Meow!!".format(self.name))  **cat = Cat('Felix') dog = Dog('Fred')  for pet in [cat, dog]:  pet.speak();** |

#### ABSTRACT CLASS

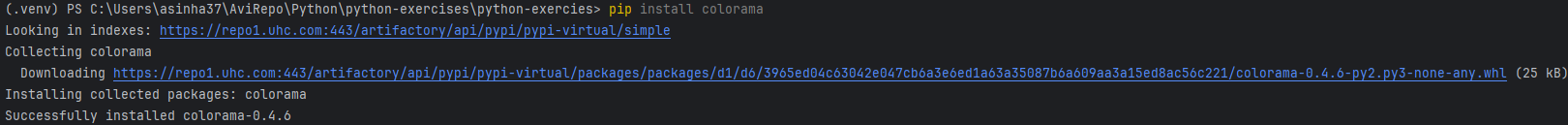
## MODULES AND PACKAGES

### PIP INSTALL AND PyPi

* PiPy(Python Package Index) is a central repository for open source third party Python Package(like npm)
* PyPI is a platform where developers can publish and distribute their Python packages for others to easily install and use in their own projects.

#### INSTALLING PACKAGES FROM PIPY

|  |  |
| --- | --- |
| TO INSTALL PACKAGES FROM PyPi | pip install requests |



### WRITING OWN PACKAGES AND MODULES

* Modules and packages are used to organize and structure code.
* A module is a single file containing Python code, while a package is a collection of modules organized in a directory hierarchy. Modules and packages allow for code reuse, modularity, and separation of concerns.

#### MODULES

* To create our own module, we need to create a new Python file with a `.py` extension.

|  |  |
| --- | --- |
| **CREATING MODULES** | |
| * Let's say we want to create a module called `**my\_module.py**`. * Create a file name **my\_module.py** * Inside this file, we can define functions, classes, or variables. | **File Name : my\_module.py**  def greet(name):  print(f"Hello, {name}!")    class MyMath:  def square(self, num):  return num \*\* 2 |
| **USING MODULES** | |
| * Once we created the module, we can import it into another Python script and use its functions or classes. * Here's an example of how to import and use the `greet` function from `my\_module`: | import my\_module    my\_module.greet("Alice") # Output: Hello, Alice! |

#### PACKAGES

* To create a package, we need to organize our modules in a directory structure.
* **The directory should contain a special file called `\_\_init\_\_.py`, which can be an empty file or can contain initialization code.**

EXAMPLE

Let's create a package called `my\_package` with two modules: `module1.py` and `module2.py`.

|  |  |
| --- | --- |
| **DIRECTORY STRUCTURE** | my\_package/  \_\_init\_\_.py  module1.py  module2.py |
| **MODULE 1(MODULE1.PY)** | def func1():  print("This is function 1")    def func2():  print("This is function 2") |
| **MODULE 2(MODULE2.PY)** | def func3():  print("This is function 3")    def func4():  print("This is function 4") |
| **USING PACKAGE** | We can then import and use the functions from the package and its modules in your code:  import my\_package.module1  import my\_package.module2    my\_package.module1.func1() # Output: This is function 1  my\_package.module2.func3() # Output: This is function 3  ***OR -*** *Alternatively, we can use the `from ... import` syntax to import specific functions directly*  from my\_package.module1 import func1  from my\_package.module2 import func3    func1() # Output: This is function 1  func3() # Output: This is function 3 |

##### SUBPACKAGES

|  |  |  |
| --- | --- | --- |
| **DIRECTORY STRUCTURE** | my\_package/  \_\_init\_\_.py  subpackage1/  \_\_init\_\_.py  module1.py  subpackage2/  \_\_init\_\_.py  module2.py |  |
| **module1.py** | def func1():  print("This is function 1 in subpackage1") def func2():  print("This is function 2 in subpackage1") | |
| **module2.py** | def func3():  print("This is function 3 in subpackage2") def func4():  print("This is function 4 in subpackage2") | |
| **USING PACKAGE / SUBPACKAGES** | from my\_package.sub\_package1 import module1 from my\_package.sub\_package2 import module2  module1.func1() module1.func2() module2.func3() module2.func4() | |

## \_\_name\_\_ and \_\_main\_\_

* Sometimes when you are importing from a module, you would like to know whether

a modules function is being used as an import, or if you are using the original

.py file of that module. In this case we can use the:

if \_\_name\_\_ == "\_\_main\_\_":

line to determine this. For example:

When your script is run by passing it as a command to the Python interpreter:

python myscript.py

all of the code that is at indentation level 0 gets executed. Functions and

classes that are defined are, well, defined, but none of their code gets ran.

Unlike other languages, there's no main() function that gets run automatically

- the main() function is implicitly all the code at the top level.

In this case, the top-level code is an if block. \_\_name\_\_ is a built-in variable

which evaluate to the name of the current module. However, if a module is being

run directly (as in myscript.py above), then \_\_name\_\_ instead is set to the

string "\_\_main\_\_". Thus, you can test whether your script is being run directly

or being imported by something else by testing

if \_\_name\_\_ == "\_\_main\_\_":

...

If that code is being imported into another module, the various function and

class definitions will be imported, but the main() code won't get run. As a

basic example, consider the following two scripts:

# file one.py

def func():

print("func() in one.py")

print("top-level in one.py")

if \_\_name\_\_ == "\_\_main\_\_":

print("one.py is being run directly")

else:

print("one.py is being imported into another module")

and then:

# file two.py

import one

print("top-level in two.py")

one.func()

if \_\_name\_\_ == "\_\_main\_\_":

print("two.py is being run directly")

else:

print("two.py is being imported into another module")

Now, if you invoke the interpreter as

python one.py

The output will be

top-level in one.py

one.py is being run directly

If you run two.py instead:

python two.py

You get

top-level in one.py

one.py is being imported into another module

top-level in two.py

func() in one.py

two.py is being run directly

Thus, when module one gets loaded, its \_\_name\_\_ equals "one" instead of \_\_main\_\_.

## ERROR HANDLING

* Error handling allows us to gracefully handle and manage exceptions or errors that may occur during the execution of the code.

### MECHANISMS FOR ERROR HANDLING

|  |  |
| --- | --- |
| **Try-Except**:   * The `try-except` block is used to catch and handle specific exceptions. * We enclose the code that may raise an exception in the `try` block, and then specify the exception type(s) we want to handle in the `except` block. | try:  # Code that may raise an exception  result = 10 / 0  except ZeroDivisionError:  # Code to handle the ZeroDivisionError  print("Error: Division by zero occurred") |
| **Try-Except-Else**:   * The `try-except-else` block allows us to specify code that should be executed if no exception occurs. | try:  # Code that may raise an exception  result = 10 / 2  except ZeroDivisionError:  # Code to handle the ZeroDivisionError  print("Error: Division by zero occurred")  **else**:  # Code to execute if no exception occurs  print("Result:", result) |
| **Try-Finally**:   * The `try-finally` block ensures that a specified block of code is always executed, regardless of whether an exception occurs or not. This is useful for performing cleanup operations. | try:  # Code that may raise an exception  file = open("my\_file.txt", "r")  # Code to read from the file  finally:  # Code to always execute, like closing the file  file.close() |
| **Raise**:   * We can use the `**raise**` statement to manually raise exceptions in the code. * This is useful when we want to explicitly handle specific situations or create custom exceptions. | age = -1  if age < 0:  raise ValueError("Age cannot be negative") |

## DECORATORS

### ASSIGNING FUNCTION TO VARIABLE

* Functions can be assigned to variables, just like any other object. Assigning a function to a variable allows us to refer to the function using that variable name and call the function through the variable.

|  |  |
| --- | --- |
| * In this example, the `greet` function is defined, which prints "Hello, world!". The `my\_function` variable is then assigned the value of the `greet` function. When `my\_function()` is called, it behaves the same as calling `greet()`, printing "Hello, world!". | def greet():  print("Hello, world!")  my\_function = greet  my\_function() |
| * We can also pass the function assigned to a variable as an argument to another function or return it from a function: * In this example, the `wrapper` function takes a function as an argument and executes it. The `my\_function` variable, which holds the `greet` function, is passed as an argument to `wrapper`. When `wrapper(my\_function)` is called, it first prints "Before function execution", then calls `my\_function()` (which is equivalent to calling `greet()`), and finally prints "After function execution". | def greet():  print("Hello, world!")    def wrapper(func):  print("Before function execution")  func()  print("After function execution")    my\_function = greet  wrapper(my\_function) |
| * Assigning functions to variables can be useful in scenarios like   + Where we want to pass functions as arguments   + Return functions from other functions   + Dynamically choose which function to call based on certain conditions.   + It provides flexibility and allows for more advanced programming techniques like higher-order functions and callbacks. |  |

#### SCENARIOS

|  |  |
| --- | --- |
| **PASSING FUNCTIONS AS ARGUMENTS**  def apply\_operation(func, num):  return func(num)    def square(x):  return x \*\* 2    def cube(x):  return x \*\* 3    **result = apply\_operation(square, 5) # Passes the square** function as an argument  print(result) # Output: 25    **result = apply\_operation(cube, 3) # Passes the cube** function as an argument  print(result) # Output: 27 | **RETURNING FUNCTIONS FROM OTHER FUNCTIONS**  def get\_operation(operation):  if operation == "add":  def add(a, b):  return a + b  return add  elif operation == "subtract":  def subtract(a, b):  return a - b  return subtract    operation\_func = get\_operation("add")  result = operation\_func(3, 4) # Calls the returned add function  print(result) # Output: 7    operation\_func = get\_operation("subtract")  result = operation\_func(8, 5) # Calls the returned subtract function  print(result) # Output: 3 |
| **DYNAMICALLY CHOOSING WHICH FUNCTION TO CALL**  def fast\_function():  print("Fast function called")    def slow\_function():  print("Slow function called")  mode = "fast"  if mode == "fast":  function\_to\_call = fast\_function  else:  function\_to\_call = slow\_function  **function\_to\_call()** # Calls the chosen function based on the mode | |

#### DECORATORS

* A decorator is a design pattern in Python that allows us to modify or enhance the behavior of functions or classes without directly changing their source code.
* Decorators are functions that wrap around other functions or classes to provide additional functionality. They are denoted by the `@decorator` syntax, where `decorator` is a function that takes a function or class as input and returns a modified version of it.
* When a function or class is decorated, it is essentially passed as an argument to the decorator function, and the decorator function returns a new function or class that incorporates the modifications or enhancements.

##### PURPOSE OF DECORATORS

* **ADDING FUNCTIONALITY**:
  + To add new behavior or modify the existing behavior of functions or classes.
  + For example, we can use decorators to add logging, authentication, caching, or error handling to functions.
* **MODIFYING BEHAVIOR**:
  + To change the behavior of functions or classes by wrapping them with additional code.
  + This can be useful for tasks like input validation, parameter manipulation, or altering the return value.
* **CODE ORGANIZATION**:
  + Decorators can help organize and separate concerns in your code.
  + By separating cross-cutting concerns into decorators, we can keep the core functions or classes clean and focused on their primary responsibilities.
* **REUSABILITY**:
  + Decorators promote code reusability by allowing us to apply the same modifications or enhancements to multiple functions or classes. You can define a decorator function once and use it on multiple functions or classes throughout the codebase.

##### EXAMPLES

|  |  |
| --- | --- |
| * In this example, we define a decorator function called **uppercase\_decorator**. * It takes a function as input (func) and defines an inner function called wrapper. The wrapper function wraps the execution of the original function (func) and modifies its result by converting it to uppercase. Finally, the wrapper function is returned. * The @uppercase\_decorator syntax is used to apply the uppercase\_decorator to the say\_hello function. This means that the say\_hello function is wrapped with the functionality provided by the uppercase\_decorator. * When say\_hello() is called, it executes the code inside the wrapper function. The wrapper function first calls the original say\_hello function, which returns the string "Hello, world!". Then, it converts the original result to uppercase and returns the modified result, which is then printed. | def uppercase\_decorator(func):  def wrapper():  original\_result = func()  modified\_result = original\_result.upper()  return modified\_result  return wrapper    @uppercase\_decorator  def say\_hello():  return "Hello, world!"    print(say\_hello()) # Output: "HELLO, WORLD!" |
| * In this example, the `log\_decorator` function is a decorator that wraps around the `add` function. When the `add` function is called, the decorator function adds logging statements before and after the function call. The output of the program shows the log messages, indicating the function call and completion. * Decorators are a powerful feature in Python that provide flexibility, code organization, and modularity. They enable you to enhance and modify the behavior of functions and classes without directly modifying their source code, leading to cleaner and more maintainable code. | def log\_decorator(func):  def wrapper(\*args, \*\*kwargs):  print(f"Calling function: {func.\_\_name\_\_}")  result = func(\*args, \*\*kwargs)  print(f"Function {func.\_\_name\_\_} completed")  return result  return wrapper    @log\_decorator  def add(a, b):  return a + b    result = add(3, 4) # Output: Calling function: add, Function add completed  print(result) # Output: 7 |
| **CLASS DECORATORS**   * When decorators applied to a class, a decorator can modify the class definition or add additional functionality to the class. | def add\_perimeter(cls):  def calculate\_perimeter(self):  return 2 \* 3.14 \* self.radius   cls.calculate\_perimeter = calculate\_perimeter  return cls @add\_perimeter class Circle:  def \_\_init\_\_(self, radius):  self.radius = radius   def calculate\_area(self):  return 3.14 \* self.radius \*\* 2   circle = Circle(5) **print(circle.calculate\_area()) print(circle.calculate\_perimeter())** |

## ADVANCED MODULES

### COLLECTIONS MODULE

* The `collections` module in Python provides additional data structures and utilities that are not available in the built-in data types.
* It includes several specialized container datatypes, **such as `namedtuple`, `deque`, `Counter`, `defaultdict`, and `OrderedDict`, which offer enhanced functionality compared to the standard data types**.

#### COMMONLY USED CLASSES IN THE `COLLECTIONS` MODULE

##### Counter

* It is a dict subclass that counts the occurrences of elements in a collection.
* It provides a convenient way to count elements and can be useful for tasks such as frequency analysis.

|  |  |
| --- | --- |
| EXAMPLE 1 | **from collections import Counter,**  # Counter example  c = Counter(['a', 'b', 'a', 'c', 'b', 'a'])  print(c) **# Output: Counter({'a': 3, 'b': 2, 'c': 1})** |
| EXAMPLE- 2 | sentence = "This is a very very very very long sentence" count = Counter(sentence.split()) print(count)  **Counter({'very': 4, 'This': 1, 'is': 1, 'a': 1, 'long': 1, 'sentence': 1})** |
| EXAMPLE- 3 | letters = "aaaaabbbbbccccccdddddd" count = Counter(letters) print(count)  **Counter({'c': 6, 'd': 6, 'a': 5, 'b': 5})** |

##### COUNTER METHODS

|  |  |
| --- | --- |
| **most\_common()**   * The most\_common() method is a convenient way to find the most common elements in a list | from collections import Counter num\_list = [1,1,1,1,2,2,2,2, 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c', 'c', 'c'] count= Counter(num\_list) print(count.most\_common())  **OUTPUT - [('c', 5), (1, 4), (2, 4), ('b', 3), ('a', 2)]** |
| * The parameter to most\_common(2) will output the top 2 most common | print(count.most\_common(2))  **OUTPUT - [('c', 5), (1, 4)** |

\

|  |
| --- |
| from collections import Counter  # Creating a Counter object  c = Counter([1, 2, 2, 3, 3, 3, 4, 4, 4, 4]) |
| # Total of all counts  print(sum(c.values())) # Output: 10 |
| # Reset all counts  c.clear()  print(c) # Output: Counter() |
| # List unique elements  print(list(c)) # Output: [] |
| **# Convert to a set :** print(set(c)) # Output: {1, 2, 3, 4} |
| # Convert to a regular dictionary  print(dict(c)) # Output: {} |
| # Convert to a list of (elem, cnt) pairs  print(c.items()) # Output: dict\_items([]) |
| # Convert from a list of (elem, cnt) pairs  list\_of\_pairs = [('a', 2), ('b', 3), ('c', 1)]  c = Counter(dict(list\_of\_pairs))  print(c) # Output: Counter({'b': 3, 'a': 2, 'c': 1}) |
| # N least common elements  n = 2  print(c.most\_common()[:-n-1:-1]) # Output: [('c', 1), ('a', 2)] |
| # Remove zero and negative counts  c += Counter()  print(c) # Output: Counter({'b': 3, 'a': 2, 'c': 1}) |

##### namedtuple

* It is a factory function that creates tuple subclasses with named fields.
* It allows us to access tuple elements by name, making the code more readable and self-explanatory.

|  |  |
| --- | --- |
| EXAMPLE | from collections import namedtuple  # namedtuple example  Point = namedtuple('Point', ['x', 'y'])  p = Point(2, 3)  print(p.x, p.y) # Output: 2 3 |

##### deque

* It is a double-ended queue that allows efficient appending and popping of elements from both ends.
* It provides fast operations for inserting and removing elements from the beginning or end of the queue.

|  |  |
| --- | --- |
| EXAMPLE | from collections import deque  # deque example  d = deque([1, 2, 3])  d.append(4)  d.appendleft(0)  print(d) # Output: deque([0, 1, 2, 3, 4]) |

##### defaultdict

* It is a dict subclass that provides a default value for missing keys.
* It eliminates the need for checking if a key exists before accessing its value, making the code cleaner and more concise.

|  |  |
| --- | --- |
| EXAMPLE | from collections import defaultdict,  # defaultdict example  d = defaultdict(int)  d['apple'] += 1  d['banana'] += 2  print(d) # Output: defaultdict(<class 'int'>, {'apple': 1, 'banana': 2}) |

##### OrderedDict

* It is a dict subclass that remembers the order of key-value pairs.
* It maintains the order in which the elements were added, making it useful in scenarios where the order of insertion matters.

|  |  |
| --- | --- |
| EXAMPLE | from collections import OrderedDict  # OrderedDict example  od = OrderedDict()  od['a'] = 1  od['b'] = 2  od['c'] = 3  print(od) # Output: OrderedDict([('a', 1), ('b', 2), ('c', 3)]) |

#### OSMODULE

* The os module in Python provides a way to interact with the operating system.
* It offers functions for performing various tasks related to files, directories, processes, environment variables

|  |
| --- |
| * **os.name:**    + Returns the name of the operating system. |
| * **os.getcwd():**    + Returns the current working directory. |
| * **os.chdir(path):**    + Changes the current working directory to the specified path. |
| * **os.listdir(path):**    + Returns a list of files and directories in the specified path.   + If no path specified, it will list the files in current directory |
| * **os.mkdir(path):**   + Creates a new directory with the specified path. |
| * **os.remove(path):**    + Deletes the file at the specified path. |
| * **os.rmdir(path):**    + Removes the empty directory at the specified path. |
| * **os.path.join(path, \*paths):**    + Joins one or more paths to create a single path. |
| * **os.path.exists(path):**    + Returns True if the file or directory at the specified path exists. |
| * **os.path.isfile(path):**   + Returns True if the specified path points to a file. |
| * **os.path.isdir(path):**    + Returns True if the specified path points to a directory. |
| * **os.environ:**    + A dictionary containing the environment variables. |
| * **os.getenv(var\_name):**    + Returns the value of the environment variable with the specified var\_name. |
| * **os.system(command):**    + Executes the specified command in the system shell. |

#### DATETIME

#### MATH AND RANDOM

#### TIMEIT

#### REGULAR EXPRESSION

#### UNZIP AND ZIP MODULE

## WEB SCRAPING

* Web scraping is the process of extracting data from websites by using automated methods or tools. It involves
  + Fetching web pages,
  + Parsing the HTML or XML content
  + Extracting the desired information.

Web scraping allows us to gather data from multiple web pages and websites, which can then be used for various purposes, such as data analysis, research, or building applications.

### WEB SCRAPING PROCESS

|  |
| --- |
| **INSTALL FOLLOWING PACKAGE** |
| **Step 1: Fetching web pages**:   * Web scraping starts with sending HTTP requests to the target website's server to retrieve the HTML content of the desired web page. * This can be done using programming libraries like **requests or urllib** in Python.   **pip install requests** |
| **Step 2: Parsing HTML**:   * Once the HTML content is obtained, it needs to be parsed to extract the relevant data. * This is typically done using parsing libraries like BeautifulSoup or lxml in Python. * These libraries provide convenient methods to navigate and search through the HTML structure.   **pip install lxml**  **pip install bs4** |
| **Step 3: Extracting data**:   * After parsing the HTML, we can use the libraries' methods to extract the desired data from specific HTML elements or attributes. * This can include text, links, images, tables, or any other relevant information present on the web page. |
| **Step 4: Data processing and storage**:   * Once the data is extracted, we can process and manipulate it as per your requirements. * This may involve cleaning the data, performing calculations, or transforming it into a suitable format.   We can store the scraped data in a file or a database for further analysis or use in our applications. |

#### GRABBING IMAGES

* Let’s so web scraping of <https://example.com/>

|  |  |
| --- | --- |
| **GRABBING TITLE**  The selected code performs a simple web scraping task using the `requests` and `BeautifulSoup` libraries  **IMPORT LIBRARIES**   * `requests`: Used to send HTTP requests. * `bs4` (BeautifulSoup): Used to parse HTML and XML documents.   **SEND HTTP GET REQUEST**   * Sends a GET request to `http://example.com/` and stores the response in the `result` variable.   **PARSE HTML CONTENT**   * Parses the HTML content of the response using BeautifulSoup with the `lxml` parser.   **EXTRACT AND PRINT TITLE**   * Selects the `<title>` element from the parsed HTML. * Extracts the text content of the first `<title>` element and prints it. | **import requests, bs4**  result = requests.get('http://example.com/')  soup = bs4.BeautifulSoup(result.text, 'lxml')  print(**soup.select('title')[0].getText())**   * This code will output the title of the webpage located at `http://example.com/`. * “select” function excepts “selector” , similar to JQuery selector |

A screenshot of a computer

Description automatically generated

|  |  |
| --- | --- |
| import requests, bs4 basePath = "https://www.aarpmedicareplans.com" ; result = requests.get(basePath) image =soup.select('img.loadOnlyDesktop')[0] imgSrc = image['data-src'] downloaded\_image\_link = requests.get(basePath + imgSrc) **downloaded\_image = open('downloaded\_image.jpg', 'wb')** downloaded\_image.write(downloaded\_image\_link.content) downloaded\_image.close() | **EXTRACTING IMAGE AND DOWNLOAD**   * ”**wb**” 🡪Creates a new file called 'downloaded\_image.jpg' in write binary mode ('wb') using open() |

## WORKING WITH PDFs AND SPREADSHEETS