Contents

[PYTHON 4](#_Toc186624496)

[PYPI (PYTHON PACKAGE INDEX): 4](#_Toc186624497)

[ANACONDA 4](#_Toc186624498)

[RUNNING PYTHON FILE FROM COMMAND LINE (REPL – READ EVALUATE PRINT LOOP) 4](#_Toc186624499)

[VARIABLE 5](#_Toc186624500)

[PYTHON DATA TYPES 5](#_Toc186624501)

[OPERATORS 5](#_Toc186624502)

[TYPE CONVERSION 6](#_Toc186624503)

[TYPE CHECKING 6](#_Toc186624504)

[NUMBERS 7](#_Toc186624505)

[DATATYPES 7](#_Toc186624506)

[STRINGS 7](#_Toc186624507)

[ITERABLES 11](#_Toc186624508)

[LISTS 11](#_Toc186624509)

[DICTIONARIES 13](#_Toc186624510)

[TUPLES 15](#_Toc186624511)

[SETS 15](#_Toc186624512)

[SET AND DICTIONARY COMPREHENSION 16](#_Toc186624513)

[ZIP FUNCTION 17](#_Toc186624514)

[ENUMERATE 18](#_Toc186624515)

[BOOLEANS 18](#_Toc186624516)

[VARIABLE SCOPE 19](#_Toc186624517)

[DESTRUCTURING 20](#_Toc186624518)

[STANDARD DESTRUCTURING ASSIGNMENTS 20](#_Toc186624519)

[DESTRUCTURING DICTIONARIES IN PYTHON 20](#_Toc186624520)

[DESTRUCTURING IN FOR LOOPS 20](#_Toc186624521)

[USING \* TO COLLECT VALUES 20](#_Toc186624522)

[UNDERSCORE VARIABLES 21](#_Toc186624523)

[COMMENTS AND DOCSTRING 21](#_Toc186624524)

[COMMENTS 21](#_Toc186624525)

[DOCSTRINGS 21](#_Toc186624526)

[STATEMENTS 22](#_Toc186624527)

[IF-ELSE 22](#_Toc186624528)

[NESTED IF-ELSE 22](#_Toc186624529)

[ELIF 23](#_Toc186624530)

[is KEYWORD 23](#_Toc186624531)

[Is VERSUS == 23](#_Toc186624532)

[in KEYWORD 24](#_Toc186624533)

[LOOPS 24](#_Toc186624534)

[FOR LOOP 24](#_Toc186624535)

[WHILE LOOP 24](#_Toc186624536)

[ELSE IN LOOP 25](#_Toc186624537)

[EXAMPLES 25](#_Toc186624538)

[SOME USEFUL OPERATORS AND FUNCTIONS 25](#_Toc186624539)

[RANGE 25](#_Toc186624540)

[SORT 26](#_Toc186624541)

[FUNCTIONS 26](#_Toc186624542)

[DEFAULT VALUE OF FUNCTION PARAMETERS 26](#_Toc186624543)

[UNPACKING ARGUMENT 27](#_Toc186624544)

[LAMBDA FUNCTIONS, MAP AND FILTERS 28](#_Toc186624545)

[FIRST CLASS FUNCTION 28](#_Toc186624546)

[MAP 30](#_Toc186624547)

[FILTERS 30](#_Toc186624548)

[NESTED STATEMENT AND SCOPE 30](#_Toc186624549)

[INTERACTIVITY – USER INPUTS 31](#_Toc186624550)

[OOPS IN PYTHON 32](#_Toc186624551)

[CLASSES AND OBJECTS 32](#_Toc186624552)

[CLASS OBJECT ATTRIBUTES 32](#_Toc186624553)

[\_\_repr\_\_ and \_\_str\_\_ METHODS 32](#_Toc186624554)

[CLASS AND STATIC METHODS 33](#_Toc186624555)

[INHERITANCE 34](#_Toc186624556)

[COMPOSITION 35](#_Toc186624557)

[POLYMORPHISM 35](#_Toc186624558)

[MULTIPLE INHERITANCE 36](#_Toc186624559)

[ABSTRACT CLASSES 37](#_Toc186624560)

[TYPE HINTING 38](#_Toc186624561)

[MODULES AND PACKAGES 38](#_Toc186624562)

[PIP INSTALL AND PyPi 38](#_Toc186624563)

[WRITING OWN PACKAGES AND MODULES 38](#_Toc186624564)

[\_\_name\_\_ and \_\_main\_\_ 40](#_Toc186624565)

[ERROR HANDLING 42](#_Toc186624566)

[MECHANISMS FOR ERROR HANDLING 42](#_Toc186624567)

[RAISING ERRORS 42](#_Toc186624568)

[CUSTOM ERRORS 42](#_Toc186624569)

[DECORATORS 43](#_Toc186624570)

[ASSIGNING FUNCTION TO VARIABLE 43](#_Toc186624571)

[ADVANCED MODULES 46](#_Toc186624572)

[COLLECTIONS MODULE 46](#_Toc186624573)

[DATETIME 49](#_Toc186624574)

[MATH AND RANDOM 49](#_Toc186624575)

[TIMEIT 49](#_Toc186624576)

[REGULAR EXPRESSION 49](#_Toc186624577)

[UNZIP AND ZIP MODULE 49](#_Toc186624578)

[WEB SCRAPING 49](#_Toc186624579)

[WEB SCRAPING PROCESS 49](#_Toc186624580)

[FILES 50](#_Toc186624581)

[I/O FILES 50](#_Toc186624582)

[WORKING WITH SPREADSHEETS(CSV) 52](#_Toc186624583)

[WORKING WITH PDFs 54](#_Toc186624584)

[UNIT TEST 55](#_Toc186624585)

[LINTING - PYLINT 55](#_Toc186624586)

[TESTING LIBRARY 57](#_Toc186624587)

# 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)

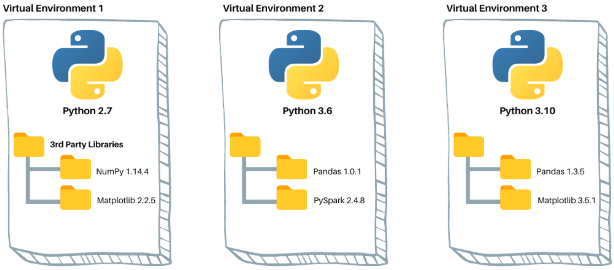
* Enter the “**python**” command to in terminal to start the Python **REPL**

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

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

## VIRTUAL ENVIRONMENT

A virtual environment is a self-contained directory that contains a specific version of Python interpreter along with its installed packages. It allows you to isolate Python projects and their dependencies, ensuring that each project can have its own set of packages without interfering with each other.



**VIRTUAL ENVIRONMENTS ARE USEFUL IN SEVERAL SCENARIOS:**

1. **Dependency Management**:
   1. If we are working on multiple projects that require different versions of packages or modules, virtual environments allow we to keep them separate and avoid conflicts.
2. **Project Isolation**:
   1. Virtual environments provide a clean and isolated environment for each project, making it easier to manage dependencies and avoid compatibility issues.
3. **Reproducibility**:
   1. By using virtual environments, we can easily share our project with others and ensure that they can reproduce the same environment and run your code without any issues.

|  |
| --- |
| To create a virtual environment, we can use tools like **`venv`, `virtualenv`, or `conda`.** |

### SETTING UP VENV IN WINDOWS (USING PIP)

Step 1: Create a directory where the virtual environment needs to be set up

Step 2: Run the following command in the cmd line

|  |  |
| --- | --- |
| CREATE THE VIRTUAL ENVIRONMENT USING venv   * Virtual environments are created using tools like **virtualenv** or the built-in **venv module** in Python. * These tools create a separate directory with its own Python executable, libraries, and scripts. * This command uses the built-in venv module in Python to create a new virtual environment * This is the recommended way to create virtual environments in Python 3.   **python -m venv <virtual\_env\_name>**  **python -m venv flaskenv** | |
| CREATE THE VIRTUAL ENVIRONMENT USING **virtualenv**   * This command uses the third-party package virtualenv to create a new virtual environment named "**venv**". * It is executed directly from the command line, without invoking the Python interpreter. * This method can be used in both Python 2 and Python 3. | |
| Step 1: INSTALLING VIRTUALENV PACKAGE: **pip install virtualenv** | |
| Step 2: CREATE VIRTUAL ENVIRONMENT: **virtualenv <virtual\_env\_name>** | |
| ACTIVATE THE VIRTUAL ENVIRONMENT: | <**virtual\_env\_name**>\Scripts\activate.bat  flaskenv\Scripts\activate.bat |
| A screen shot of a computer  Description automatically generated | |
| * Once inside a virtual environment, we can use the Python interpreter and install packages using tools like pip. * Any packages installed in the virtual environment will only be available within that environment and won't interfere with other projects or the system. | |
| For example – Let install the “flask” package in the virtual environment (fastapiapp)  **pip install flask**   * When we execute – **pip list** command it in the virtual environment it will show the flask in installed packages list | A screenshot of a computer  Description automatically generated |
| Once we are in virtual environment – we can deactivate it too | COMMAND : **deactivate** |

***NOTE: -When we activate a virtual environment, it modifies your system's environment variables to prioritize the isolated environment over the system-wide Python installation.***

#### CONFIGURE VIRTUAL ENVIRONMENT IN PYCHARM

* Click on the interpreter (right buttom corner in the IDE) 🡪 Add New Interpreter🡪 Add local Interpreter

A screenshot of a computer program

Description automatically generated

## requirements.txt

**A `requirements.txt` file is commonly used in Python projects to specify the dependencies and their versions required for the project to run. It is useful for managing and reproducing the project's environment and avoid compatibility issues (like package.json)**

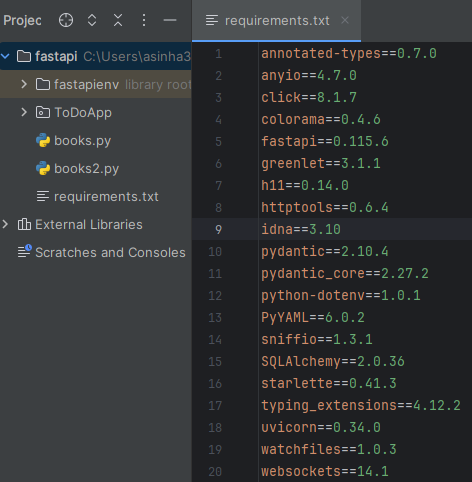
### STEPS TO CREATE requirements.txt

1. **Step 1:** When a given virtual environment is activated -  run the command **pip freeze** in terminal.

|  |  |
| --- | --- |
| The pip freeze command will output a list of installed packages and their versions in the format package==version. For example: | flask==1.1.2  requests==2.25.1  pandas==1.2.4 |

1. **Step 2:**

* Redirect the output to a text file. For example, we can use the command **pip freeze > requirements.txt** to save the output to a file named "**requirements.txt**".



1. **Step 3:**

* The resulting list of packages and versions generated by pip freeze can be used to recreate the same environment on another machine or to install the same set of packages in a different virtual environment.
* This is often done by sharing the "**requirements.txt**" file and using the **pip install -r requirements.txt** command to install the packages listed in it.

## PIPENV

* Pipenv is a tool that combines the functionality of **pip (the package installer for Python) and virtualenv (the tool for creating virtual environments)**.
* It aims to provide a more streamlined and user-friendly experience **for managing Python dependencies and virtual environments within projects.**

|  |  |
| --- | --- |
| **INSTALL PIP ENV PACKAGE** | **pip install pipenv** |
| **Type ”pipenv” command to validate** |  |

### INSTALLING DEPENDENCIES

* If we have requirements.txt already in the project . Run ”**pipenv install**” command. This will perform following

1. Install all the dependencies mentioned in requirements.txt.
2. **Creates Pipfile and Pipfile.lock files**
3. **Creates the virtual environment**

A screen shot of a computer

Description automatically generated

#### Pipfile and Pipfile.lock FILES

* The Pipfile and Pipfile.lock files are used by **Pipenv**, a Python dependency management tool, to manage project dependencies and create deterministic builds.

##### Pipfile

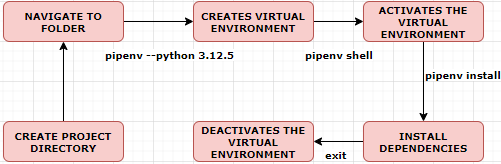
* The Pipfile is a human-readable file used to specify the dependencies and their versions for a Python project. (*It is intended to replace the traditional requirements.txt file. )*
* The Pipfile is written in **TOML** (Tom's Obvious, Minimal Language) format and contains two main sections:

|  |  |
| --- | --- |
|  | * **[packages]**: * This section lists the direct dependencies of the project. * Each package is specified with its name and optional version specifier. * **[dev-packages]**: * This optional section lists the development dependencies, which are only required for development and testing. It follows the same format as the `[packages]` section. |

##### Pipfile.lock:

* The Pipfile.lock file is automatically generated by Pipenv when dependencies are installed or updated.
* It is a lockfile that records the exact versions of all the dependencies, including their sub-dependencies.
* The purpose of the lockfile is to ensure deterministic builds, meaning that the exact same versions of dependencies are installed consistently across different environments.
* The Pipfile.lock file is automatically updated when we run commands like `**pipenv install**` or `**pipenv update**`.
* It should be committed to version control, and when other developers or deployment environments use `pipenv install`, Pipenv will install the exact same versions of packages specified in the lockfile.
* The Pipfile.lock file also includes other information such as hashes of packages to ensure integrity and security.

##### VIRTUAL ENVIRONMENT – PIP



1. **Install Pipenv**: If you haven't already, install Pipenv on your system by running pip install pipenv in your command prompt or terminal.
2. **Navigate to your project directory**: Open a command prompt or terminal and navigate to the directory where your Python project is located.
3. **Create a virtual environment**: Run the command pipenv --python 3 to create a virtual environment using the latest installed version of Python 3. If you want to use a specific version of Python, you can replace 3 with the desired version number, such as 3.8.
4. **Activate the virtual environment**: Run the command pipenv shell to activate the virtual environment. Once activated, your command prompt or terminal prompt will change, indicating that you are now working within the virtual environment.
5. **Install project dependencies**: With the virtual environment activated, you can use the pipenv install command to install project dependencies specified in the Pipfile. This command will create or update the Pipfile.lock file as well.
6. **Run Python commands**: Now you can run Python commands or scripts within the virtual environment, and Pipenv will ensure that the correct packages and versions are used.
7. **Deactivate the virtual environment**: To exit the virtual environment, simply run the command exit or close the command prompt or terminal window. This will deactivate the virtual environment and return you to your system's default Python environment.

## VARIABLE

Best practices in variable declaration

1. Declare constants in upper case. PI = 3.14
2. Declare non-constant variables in snake case. e.g. **even\_numbers =[2,4,6]**

## PYTHON DATA TYPES

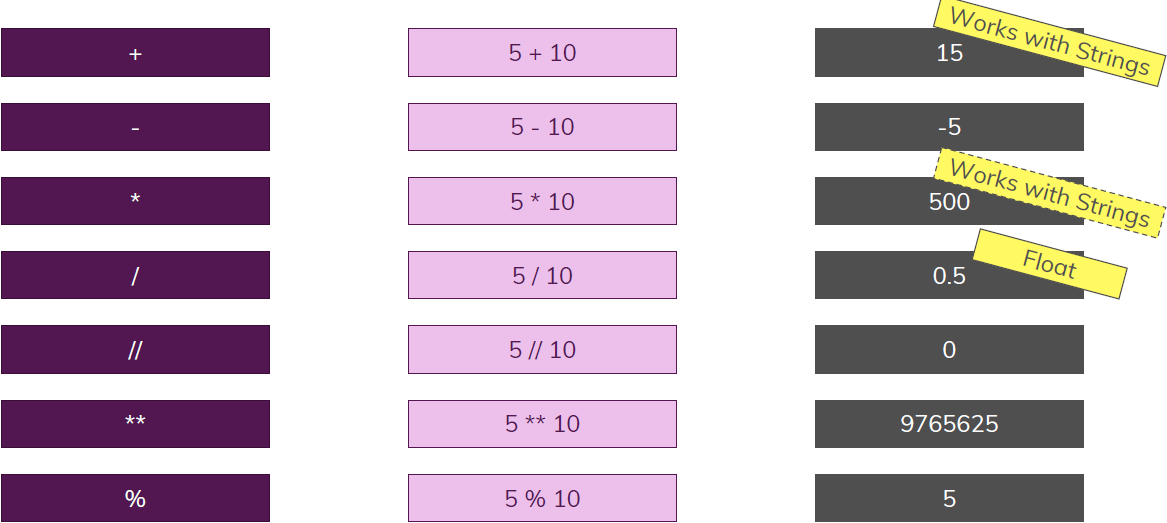
|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Example(s)** |
| **int** | Integer numbers without decimal points | x = 5 |
| **float** | Floating-point numbers with decimal points | y = 3.14 |
| **str** | Sequence of characters enclosed in single or double quotes | name = "John" |
| **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.

## NUMBERS

|  |  |
| --- | --- |
| **int(8/3)** | OUTPUT: 2  Trims the decimal points |
| **round(8/3,2)** | OUTPUT: 2.67  Round it to 2 decimal places |
| **8//3** | Integer Division - It is same as int(8/3)  **8/3= 2.6666666666666665**  **8//3 = 2** |
| **Assignment Operator** | score=0  score +=1 🡪 1  score -=1 🡪 0 |
| **2\*\*3 OR**  **pow(2,3)** | To find the power of the number |
| **hex(num)** | Hexadecimal Representation of the number |
| **bin(num** | Binary Representation of the number |
| **abs(-2)** | OUTPUT = 2(Absolute value) |
| **BODMAS** | 1+3\*4/2-1 = 6.0  Note – The division operation always returns a float even if the result is an integer e.g. (4/2 = 2.0) |

## 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

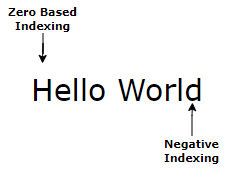
Strings are immutable. This means that once a string is created, we cannot change its individual characters. However, we can create a new string by performing operations on the original string.

|  |  |  |
| --- | --- | --- |
| 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 | |
| * **An string cannot be concatenated with non-string data types** * **The non-string data type need to be converted to string before concatenation** * **OR we can use String Formatting using f-string(template string) or “format()” method** | **Example**  age = 34  print ("Age is "+age) 🡪 Error | **SOLUTION**  age = str(34)  print ("Age is "+age) |
| **print(f”The age ={age}”)** | |
| **ESCAPING IN STRING**  message = "He said \"Python is easy\"" | **OUTPUT** - 'He said "Python is easy"' | |

#### 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! |
| **MULTILINE 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.** |
| * The multiline strings are typically used to define string literals, they can also be used as comments by placing them within the code **but not assigning them to any variable**. These multiline strings act as comments and are ignored by the Python interpreter. * In the example, the multiline string is enclosed within triple quotes and contains multiple lines of text. It serves as a comment to provide information or explanations about the code. The Python interpreter ignores this multiline string, and only the `"Hello, World!"` statement is executed. * Using multiline strings as comments can be useful when we need to include longer explanations or documentation within your code. However, it's important to note that these multiline strings are still stored in memory, so they may consume some memory resources. | | **"""**  **This is a multiline string**  **that can be used as a comment.**  **It can span multiple lines**  **without affecting the code.**  **"""**    **print("Hello, World!")** |

#### STRING SLICING AND INDEXING



##### INDEXING

**Syntax: sequence[index]**

|  |  |  |
| --- | --- | --- |
| COMMAND | OUTPUT | NOTES |
| message[0] | ‘H’ | Indexing starts at 0 for the first element |
| message[-1] | ‘d’ | Negative indexing starts at -1 for the last element |

##### SLICING

|  |  |
| --- | --- |
| **Syntax: sequence** **[start:stop:step]** | Parameters:   * start (inclusive): Index where the slice begins. * stop (exclusive): Index where the slice ends. * step (optional): Step size for selecting elements. Default Step is 1 |

|  |  |  |
| --- | --- | --- |
| COMMAND | OUTPUT | NOTES |
| message[1:3] | ‘el’ | Start from index =1 to elements before index =3 (exclude 3) |
| message[2:] | 'llo World' | Start with index |
| message[:3] | 'Hel' | Till before index= 3 |
| message[1:8:2] | 'el o' | Start from index =1 till before index= 8 in step of 2 |
| message[::-1] | 'dlroW olleH' | **Reverse String** |
| message[::-2] | 'drWolH' | Reverse String in step of 2 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **H** | **e** | **l** | **l** | **o** |  | **W** | **o** | **r** | **l** | **d** |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |

#### STRING FORMATTING

##### TEMPLATE STRING

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

|  |  |  |
| --- | --- | --- |
| **BASIC SYNTAX** | formatted\_string = "Hello, {}!"  message = formatted\_string. .format(value) | |
| * “formatted\_string” is a string template which can be re-used with multiple values * 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.

###### STRING FORMATING OPERATOR

* We can use **%s** to inject strings into the print statements. The modulo `%` is referred to as a **string formatting operator.**

|  |  |
| --- | --- |
| **STATEMENT** | **OUTPUT** |
| print("I'm going %s to inject here." % **'something'**) | I'm going something to inject here. |
| print("I'm going %s to inject here %s." % ('something','as well!')) | I'm going something to inject here as well!. |
| x, y = 'some', 'more'  print("I'm going to inject %s text here, and %s text here."%(x,y)) | I'm going to inject some text here, and more text here. |
| * The `%s` operator converts whatever it sees into a string, including integers and floats. * The `%d` operator converts numbers to integers first, without rounding. Note the difference below:   **print('I wrote %s programs today.' %3.75)**  **print('I wrote %d programs today.' %3.75)** | I wrote 3.75 programs today.  I wrote 3 programs today. |

###### PADDING AND PRECISION OF FLOATING POINT NUMBERS

**Floating point numbers use the format %5.2f**

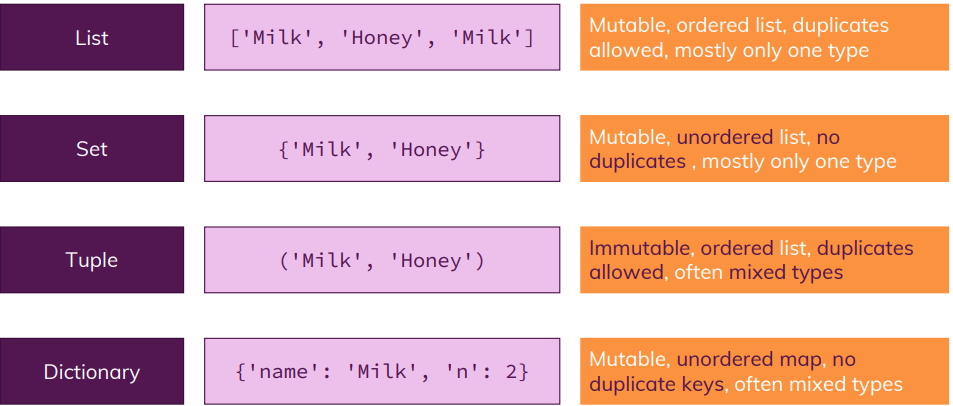
* 5 would be the minimum number of characters the string should contain; these may be padded with whitespace if the entire number does not have this many digits.
* .2f stands for how many numbers to show past the decimal point.

|  |  |
| --- | --- |
| print('Floating point numbers: %5.2f' %(13.144)) | Floating point numbers: 13.14 |
| print('Floating point numbers: %1.0f' %(13.144)) | Floating point numbers: 13 |
| print('Floating point numbers: %1.5f' %(13.144)) | Floating point numbers: 13.14400 |

#### ADVANCED STRING

|  |  |
| --- | --- |
| **Capitalize**: converts the first character of a string to uppercase and all other characters to lowercase. | message="hello world"  message.capitalize()  **OUTPUT : Hello world** |
| **count(substring**): This method returns the number of occurrences of a specified substring within a string. | text = "Hello, hello, hello"  count = text.count("hello")  print(count)  **OUTPUT : 3** |
| **find(substring):** This method returns the index of the first occurrence of a specified substring within a string. If the substring is not found, it returns -1 | text = "Hello, world"  index = text.find("world")  print(index)  **OUTPUT : 7** |
|  |  |

## ITERABLES



### LISTS

* List is a data structure that holds an ordered collection of elements.
* List can have element of different data types
* **Unlike String - 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 - This remove the first occurrence of the element in the list**  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*  **LIST OF LIST**  fruits = [[1,"apple"], [2,"banana"], [3,"orange"], [4,"grape"]]  fruits.remove([3,"orange"])  though it will give an error  fruits.remove([“orange"]) 🡪 ERROR  **REMOVES THE SPECIFIC INDEX ELEMENT FROM THE LIST**  fruits = ["apple", "banana", "orange", "grape"]  poped\_element = fruits.pop(-1) **print(poped\_element) *#* grape** | | | **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 |
| **INSERTION IN LIST**  Inserts the element in at the specified Index  num\_list = [1, 2, 3, 4, 5, 6, 7]  num\_list.insert(3,9)  **OUTPUT -[1, 2, 3, 9, 4, 5, 6, 7]** |
| **SLICING**   * fruits=['apple', 'mango', 'orange'] * fruits[1:] 🡪['mango', 'orange'] * fruits[1:2] 🡪 ['mango'] * fruits[:2]) 🡪['apple', 'mango'] | | * fruits[:] 🡪['apple', 'mango', 'orange'] * fruits[-2:] 🡪['mango', 'orange']   Start counting from end and the print every thing till end   * fruits[:-2] 🡪[‘apple’]-   Start counting from end and the print every thing from beginning | |
| **CONCATENATION**  fruits=['apple', 'mango', 'orange']; dryFruits =['almond','peanuts'] print(fruits+dryFruits) *#['apple', 'mango', 'orange', 'almond', 'peanuts']* | | | |
| **SORT**  **mylist = ['d','f','a','x','b']**  **mylist.sort() -- > ['a', 'b', 'd', 'f', 'x']** | | | **REVERSE**  **mylist = ['a', 'b', 'd', 'f', 'x']**  **mylist.reverse()**  **['x', 'f', 'd', 'b', 'a']** |
| **The sort() and reverse() does not return anything(return “None”) but sort / reverse the actual list** | | | |
| **extend()** | | | |
| num\_list =[1,2,3,4]  num\_list.append([5,6,7])  print(num\_list)  **OUTPUT : [1, 2, 3, 4, [5, 6, 7]]** | num\_list =[1,2,3,4]  num\_list.extend([5,6,7])  print(num\_list)  **OUTPUT : [1, 2, 3, 4, 5, 6, 7]** | | |
| * The append() method adds the entire [5, 6, 7] list as a single element to the end of the num\_list. So, the resulting list contains [1, 2, 3, 4, [5, 6, 7]]. * The extend() method adds each element of the [5, 6, 7] list individually to the end of the num\_list. This results in the list being extended with the elements [5, 6, 7], resulting in [1, 2, 3, 4, 5, 6, 7]. | | | |

#### LIST COMPREHENSION

* List comprehension is a concise way to create lists from an existing list. 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(cannot be sorted)** and **changeable**.
* We cannot have duplicate keys in the dictionary
* Dictionary keeps the order in which it the elements has been added

|  |  |  |
| --- | --- | --- |
| **CREATING DICTIONARY** | person = {  "name": "John",  "age": 30,  "city": "New York"  }  In dictionaries, keys can be of types like strings and integers, but keep in mind that they can also be any other hashable types, such as tuples. | |
| **ACCESSING VALUES**  print(person["name"])  print(person["age"]) | Output: "John"  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 AS LIST**  print("values",person.values()) 🡪 output dict\_values(['Alex', 27, 'New York'])  print("keys",person.keys())🡪 output : dict\_keys(['name', 'age', 'city'])  Keys & values as a list | |
| **print(person.items()) 🡪 # output - dict\_items([('name', 'Alex'), ('age', 27), ('city', 'New York')])**  **for key, value in person.items():**  **print(key, value)**   * The person.items() method returns a view object that contains the key-value pairs of the dictionary person. * **Each key-value pair is represented as a tuple** | | |
| **NOTE :**  **ITERATE OVER THE KEYS**  **When we iterate using below “for” – we are iterating over keys of dictionary** | | **my\_dict = {"key1": "value1", "key2": "value2", "key3": "value3"}**  **for key in my\_dict:**  **print(key)** |
| **ITERATE OVER THE VALUES** | | **for value in my\_dict.values():**  **print(value)** |
| **ITERATE OVER BOTH KEYS AND VALUES** | | **for key, value in my\_dict.items():**  **print(key, value)** |

#### dict() function

* The `dict()` function is a built-in function used to create a dictionary object. It can be called with various arguments to create dictionaries in different ways.

|  |  |
| --- | --- |
| **CREATING AN EMPTY DICTIONARY**   * Both statements create an empty dictionary with no key-value pairs. | empty\_dict = dict()  # or  empty\_dict = {} |
| **CREATING A DICTIONARY FROM KEY-VALUE PAIRS:** | **student = dict(name="John", age=18, grade="A")**  **O/p 🡪 {'name': 'John', 'age': 18, 'grade': 'A'}**   * The `dict()` function is used to create a dictionary called `student` with three key-value pairs. * Each keyword argument inside the `dict()` function represents a key-value pair. |
| **CREATING A DICTIONARY FROM ITERABLE OBJECTS:** | **fruits = dict([('apple', 3), ('banana', 5), ('orange', 2)])**  **o/p : {'apple': 3, 'banana': 5, 'orange': 2}**   * `dict()` function is called with a list of tuples as its argument. * Each tuple contains a key-value pair. The `dict()` function then creates a dictionary named `fruits` based on these pairs. |
| **CREATING A DICTIONARY USING** **zip()** | **keys = ['name', 'age', 'city']**  **values = ['Alice', 25, 'New York']**  **person = dict(zip(keys, values))**  **O/P 🡪{'name': 'Alice', 'age': 25, 'city': 'New York'}**   * The `zip()` function is used to combine the `keys` and `values` lists into a single iterable, and then the `dict()` function is called to create a dictionary named `person` with the corresponding key-value pairs. |

#### UNPACKING DICTIONARY

* Dictionary unpacking in Python allows us to unpack the key-value pairs of a dictionary into variables or use them directly as arguments in a function. It is performed using the \*\* operator.

|  |  |
| --- | --- |
| **UNPACKING INTO VARIABLES - U**npack a dictionary into individual variables.  my\_dict = {"name": "Alice", "age": 25}  name, age = my\_dict.values()  print(name) # Output: Alice  print(age) # Output: 25 | **UNPACKING IN FUNCTION CALLS - Unpack** dictionary unpacking to pass key-value pairs as arguments to a function.  def greet(name, age):      return f"Hello, {name}! You are {age} years old."  person = {"name": "Alice", "age": 25}  print(greet**(\*\*person**)) # Output: Hello, Alice! You are 25 years old. |
| **MERGING DICTIONARIES - U**npack multiple dictionaries to merge them.  dict1 = {"a": 1, "b": 2}  dict2 = {"b": 3, "c": 4}  merged = {\*\*dict1, \*\*dict2} # Dict2 values will override dict1 if keys overlap  print(merged) # Output: {'a': 1, 'b': 3, 'c': 4} | **USING UNPACKING IN LOOPS - U**npacking within loops to work with keys and values directly.  my\_dict = {"x": 10, "y": 20}  for key, value in my\_dict.items():      print(f"{key} = {value}")  # Output:  # x = 10  # y = 20 |

### TUPLES

* A tuple is an **immutable collection** of **ordered** elements**. Hence we cannot add or remove element in the tuple**
* It is like a list, but unlike lists, tuples cannot be modified once created. i,e element cannot be added or removed from tuble
* Tuples are created using parentheses () or the tuple() constructor.

|  |  |
| --- | --- |
| **CREATING A TUPLE**  fruits = ("apple", "banana", "orange", "grape")  friend = “Alex”, 🡨 this also will be treated as tuple | **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) | **INDEX OF AN ELEMENT IN TUPLE**  letters = ('a','b','c','d') print(letters.index('c')) #output 2 |
| **TUPLE PACKING AND UNPACKING**  name = "John"  age = 30  person = (name, age)  print(person) # Output: ("John", 30)  ## **UNPACKING**  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 |
| **INDEXING IN TUPLE**   * In a tuple individual elements can be accessed by using their index. * We can use positive or negative indices. Positive indices count from the beginning of the tuple, while negative indices count from the end.   **my\_tuple = ("apple", "banana", "cherry", "date")**  **print(my\_tuple[0]) # Output: apple**  **print(my\_tuple[2]) # Output: cherry**  **print(my\_tuple[-1]) # Output: date** |
| **SLICING:**   * Slicing is done using the colon (:) operator. The slice notation is **start:end:step**, where start is the starting index, end is the ending index (exclusive), and step is the step size between elements.   **my\_tuple = ("apple", "banana", "cherry", "date")**  **print(my\_tuple[1:3]) # Output: ('banana', 'cherry')**  **print(my\_tuple[:2]) # Output: ('apple', 'banana')**  **print(my\_tuple[1:]) # Output: ('banana', 'cherry', 'date')**  **print(my\_tuple[::2]) # Output: ('apple', 'cherry')** |

* There is a difference between a tuple with a single element and a tuple without the comma.

|  |  |
| --- | --- |
| * `my\_tuple = ("apple",)`: This is a tuple with a single element. The trailing comma after "apple" distinguishes it from a regular string. **The comma is necessary to indicate that it is a tuple with one element. Without the comma, it would be treated as a string.** * `my\_tuple = ("apple")`: **This is not a tuple but a string. Parentheses alone do not make it a tuple. In this case, "apple" is treated as a string literal.** | my\_tuple = ("apple",)  print(type(my\_tuple)) # Output: <class 'tuple'>    my\_tuple = ("apple")  print(type(my\_tuple)) # Output: <class 'str'> |

### SETS

* **Set is an unordered collection. i.e if we add element to set – it will be added at random location within the set**
* **Set always has 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  my\_set.clear() # Removes all the 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) 🡪 **{1,2,3,4,5**}  **# RETURNS A SET CONTAINING COMMON ELEMENTS FROM BOTH SETS**  intersection\_set = set1.**intersection**(set2) 🡪 **{3}**  **# RETURNS A SET CONTAINING ELEMENTS PRESENT IN SET1 BUT NOT IN SET2**  difference\_set = set1.**difference**(set2) 🡪 **{1,2}**  **# RETURNS A SET CONTAINING ELEMENTS PRESENT IN EITHER SET1 OR SET2, BUT NOT IN BOTH**  symmetric\_difference\_set = set1.**symmetric**\_difference(set2) 🡪 **{1,2,4,5}**  **SUBSET**  **s1 =set({1,2,3,4,5})**  **s2 =set({1,2})**  **s2.issubset(s1) 🡪 True**  **s1.issuperset(s2) 🡪 True** |
| **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. |

### SET AND DICTIONARY COMPREHENSION

* Both sets and dictionaries support comprehensions, which are concise ways to create these data structures from an iterable or based on certain conditions. Comprehensions allow you to create sets or dictionaries in a single line of code.

|  |  |
| --- | --- |
| **SET COMPREHENSION**   * Set comprehensions are written using curly braces **{}.** * **SYNTAX**   **{expression for item in iterable if condition}** | **squares = {num\*\*2 for num in range(1, 6)}**  **print(squares) # Output: {1, 4, 9, 16, 25}** |
| **DICTIONARY COMPREHENSION**   * Dictionary comprehensions are written using curly braces **{}** with key-value pairs separated by a colon `**:**`.   **SYNTAX**: {key\_expression: value\_expression for item in iterable if condition}  **EXAMPLE - 1**  **squares\_dict = {num: num\*\*2 for num in range(1, 6)}**  **print(squares\_dict) # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}**  **EXAMPLE – 2**  **employee =['John', 'Jane', 'Doe', 'Doy'] jobs = ['Manager', 'Supervisor', 'Clerk', 'Driver'] emp\_dict = {employee[num] : jobs[num] for num in range(len(employee))} print(emp\_dict)**  **OUTPUT - {'John': 'Manager', 'Jane': 'Supervisor', 'Doe': 'Clerk', 'Doy': 'Driver'}** | |

### ZIP FUNCTION

* The `zip()` function is a built-in function that takes multiple iterables as arguments and returns an iterator of tuples. Each tuple contains elements from the input iterables, paired together based on their positions.

**SYNTAX - zip(\*iterables)**

* **The `zip()` function can take any number of iterables as arguments**. It returns an iterator that generates tuples containing elements from the corresponding positions of the input iterables. The length of the resulting iterator is determined by the shortest input iterable.

|  |  |
| --- | --- |
| names = ["Alice", "Bob", "Charlie"]  ages = [25, 30, 35]  cities = ["New York", "London", "Paris"]    zipped = zip(names, ages, cities)    for item in zipped:  print(item)  **OUTPUT**  **('Alice', 25, 'New York')**  **('Bob', 30, 'London')**  **('Charlie', 35, 'Paris')**  list(zipped):  [('Alice', 25, 'New York'), ('Bob', 30, 'London'), ('Charlie', 35, 'Paris')] | * In this example, we have three lists: `names`, `ages`, and `cities`. We use the `zip()` function to combine these lists into a single iterator of tuples. * Each tuple contains corresponding elements from the input lists based on their positions. The resulting iterator is stored in the variable `zipped`. |
| * Note: that the `zip()` function stops iterating as soon as the shortest input iterable is exhausted. * In this example, if one of the lists (**names**)had fewer elements than the others, the resulting iterator would only contain tuples up to the length of the shortest list. | names = ["Alice", "Bob"] ages = [25, 30, 35] cities = ["New York", "London", "Paris"]  zipped = zip(names, ages, cities)  for item in zipped:  print(item)  OUTPUT :  **('Alice', 25, 'New York')**  **('Bob', 30, 'London')** |
| names = [  {"name": "Alice", "age": 25},  {"name": "Bob", "age": 35},  {"name": "Charlie", "age": 45}, ] ages = [25, 30, 35] cities = ["New York", "London", "Paris"]  zipped = zip(names, ages, cities)  for item in zipped:  print(item) | OUTPUT  ({'name': 'Alice', 'age': 25}, 25, 'New York')  ({'name': 'Bob', 'age': 35}, 30, 'London')  ({'name': 'Charlie', 'age': 45}, 35, 'Paris') |

* The `zip()` function is commonly used when we need to process multiple iterables simultaneously, such as combining data from different sources or performing parallel iteration over multiple sequences.

### 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.   **IMPORTANT**  fruits = ['apple', 'banana', 'orange']  list(enumerate(fruits)) 🡪 List of tuple🡪 **[(0, 'apple'), (1, 'banana'), (2, 'orange')]** | |
| **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. * Note **that the index starts from 0 by default, but we 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)

#### LOGICAL OPERATORS – CHAINING COMPARISON OPERATOR

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

|  |  |
| --- | --- |
| ***And***   * *`and`: Returns `True` if both operands are `True`.* * and returns the first value if it evaluates to false, otherwise it returns the second value.   ***or***   * *`or`: Returns `True` if at least one of the operands is `True`.* * or returns the first value if it evaluates to true, otherwise it returns the second value.   ***not***   * *`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 |

##### EXAMPLES

|  |  |
| --- | --- |
| and - returns the first value if it evaluates to False | Or - the first value is returned if it evaluates to True |
| >>> None and '35'  None  >>> x = []  >>> x and 12  []  >>> 10 and 20  20 | >>> None or '35'  '35'  >>> x = []  >>> x or 12  12  >>> 10 or 20  10 |
| * ` The `**and**` operator has higher precedence than the `or` operator. * The `and` operator is evaluated before the `or` operator. If `age` is greater than 18 and less than 65, the expression `age > 18 and age < 65` evaluates to `True`. If `side\_job` is `True`, the final expression `True or True` evaluates to `True`. The result will be printed as `True` if both conditions are satisfied. | age = int(input("Enter your age: "))  side\_job = True  print(age > 18 and age < 65 or side\_job) |

#### TRUTHY AND FALSY VALUES

* When a value evaluates to True we call it "Truthy". Similarly, when it evaluates to False we call it "Falsy".
* Most values in Python are "Truthy". What that means is when you pass them to the bool() function, it returns True:

|  |  |
| --- | --- |
| >>> bool(None) 🡪 False  >>> bool('35') 🡪 True  >>> bool(10) 🡪 True  >>> bool([]) 🡪 False  >>> bool([1, 2, 3]) 🡪 True | **The values that are "Falsy" are:**   * Empty strings, lists, tuples, dictionaries, and other empty iterables. * None * 0 and other zero numbers (such as 0.0). |

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`.*   **Question : Check when the list has any evens**  nums = [1,2,3,5,7]  any([num for num in nums if num %2 ==0]) 🡪 **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. |

## DESTRUCTURING

* Documentation : <https://blog.teclado.com/destructuring-in-python/>

### STANDARD DESTRUCTURING ASSIGNMENTS

|  |
| --- |
| Python, like many programming languages, allows us to assign more than one variable at a time on a single line. We just have to provide the same number of values on both sides of the assignment. For example:  **x, y = 5, 11**  Here we assign the value 5 to x, and 11 to y. The values are assigned entirely based on order, so if we switch the variable order, or the order of the values we intend to assign, we will end up with different results.  **If we try to destructure a collection with more or fewer values than we provide variables, we end up with a ValueError**. |

### DESTRUCTURING DICTIONARIES IN PYTHON

|  |  |
| --- | --- |
| my\_dict = {"name": "Bob", "age": 25}  x, y = my\_dict | x = "name" and y = "age".  That's because by default when we use a dictionary as an unidimensional collection (such as a list or tuple), what we get back are the keys. It's the same if you call list(my\_dict), you get back ["name", "age"]. |
| If you wanted to destructure the dictionary values only  **my\_dict = {"name": "Bob", "age": 25}**  **x, y = my\_dict.values() # "Bob", 25** | x = "Bob" and y = "25". |

### DESTRUCTURING IN FOR LOOPS

|  |
| --- |
| example\_list = ["A", "B", "C"]  for counter, letter in enumerate(example\_list):  print(counter, letter)  # 0 A  # 1 B  # 2 C |
| people = [  ("Bob", 42, "Mechanic"),  ("James", 24, "Artist"),  ("Harry", 32, "Lecturer")  ]  for name, age, profession in people:  print(f"Name: {name}, Age: {age}, Profession: {profession}") |
| for person in people:  print(f"Name: {person[0]}, Age: {person[1]}, Profession: {person[2]}") |

### USING \* TO COLLECT VALUES

In Python, we can use the \* operator to collect leftover values when performing a destructuring assignment. For example, we might have a list of numbers, and we want to grab the first number, and then assign the remaining numbers to a second variable:

|  |  |
| --- | --- |
| head, \*tail = [1, 2, 3, 4, 5]  print(head) # 1  print(tail) # [2, 3, 4, 5] | \*head, tail = [1, 2, 3, 4, 5]  print(head) # [1, 2, 3, 4]  print(tail) # 5 |
| head, \*middle, tail = [1, 2, 3, 4, 5]  print(head) # 1  print(middle) # [2, 3, 4]  print(tail) # 5 | first, second, third, \*rest = [1, 2, 3, 4, 5] |

## UNDERSCORE VARIABLES

* The underscore (\_) variable has several different uses depending on the context. Here are some common use cases:

|  |  |
| --- | --- |
| * **Ignoring values**: The underscore can be used as a placeholder when we want to ignore a specific value returned by a function or assigned to a variable. It indicates that the value is not of interest and can be safely ignored. | **\_, y = (1, 2) # Ignoring the first value** |
| * **Temporary variable**: In interactive Python interpreters or as a convention in code, the underscore can be used as a temporary variable to store the result of the last expression. It can be useful when you want to access the result later or perform further operations on it. For example: | result = 2 + 3  print(\_) # Prints the result of the last expression (5) |
| * Localization: In some contexts, the underscore is used as a convention for localization and internationalization. It is used to mark strings that need to be translated into different languages. | message = \_('Hello, world!') |
| * "Private" variables: Although not enforced by the language itself, a single leading underscore is often used as a convention to indicate that a variable or method is intended for internal use within a class or module. It suggests that the variable or method should not be accessed directly from outside the class or module. |  |

* It's worth noting that the double leading underscores (e.g., `\_\_variable`) have a different meaning in Python. They trigger name mangling, where the variable name is modified to avoid conflicts in subclasses. This is often used for "private" variables in the context of class inheritance. .

## 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 |

#### bool()

* The `bool` type represents Boolean values, which can be either `True` or `False`. When used in an `if` statement, the `bool` type is automatically evaluated to determine the condition of the `if` statement.

A black text on a white grid

Description automatically generated

* When a value/ variable is passed to if condition, python evaluates the value by passing it bool()

|  |  |
| --- | --- |
| 1. **False** 2. **None** 3. **0 (integer)** 4. **0.0 (float)** 5. **"" (empty string)** 6. **[] (empty list)** 7. **{} (empty dictionary)** 8. **() (empty tuple)** 9. **set() (empty set)** | * The following values are considered `False` when converted to `bool`, and all other values are considered `True`: * Any other non-zero numbers, non-empty strings, and non-empty containers are considered `True` when converted to `bool`. Therefore, when using `bool` in an `if` statement, the condition is evaluated based on whether the expression evaluates to `True` or `False`. |

### 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") |

### is KEYWORD

"is" keywork is used for identity comparison, which checks if two objects refer to the same memory location.

|  |  |
| --- | --- |
| * In this example, both x and y are assigned the value 5. Since integers are immutable objects in Python, the interpreter reuses the memory location for the integer value 5. Therefore, "x is y" evaluates to True, indicating that x and y refer to the same memory location. * It's important to note that "is" compares the identity of the objects, not their values. If you want to compare the values of objects, we should use the "==" operator instead. | x = 5  y = 5    if x is y:  print("x and y refer to the same object") |

### Is VERSUS ==

In the context of lists in Python, the "==" operator and the "is" keyword are used for different types of comparisons:

|  |  |
| --- | --- |
| * **"==" operator**: It is used for comparing the values of two lists element by element. **It checks whether the elements in two lists are equal in both value and order** * In example - the "==" operator compares the values of each element in list1 and list2. Since the elements are the same in both lists, and the order is also the same, the condition is **true** | list1 = [1, 2, 3]  list2 = [1, 2, 3]    if list1 == list2:  print("list1 and list2 are equal") |
| * The "is" keyword checks if list1 and list2 are referencing the same list object in memory. Since list2 is assigned to the same list as list1, the condition is true. | list1 = [1, 2, 3]  list2 = list1  list3 = [1, 2, 3]  print(list1 is list2) 🡪 True  print(list1 is list3) 🡪 False |

|  |
| --- |
| It's important to note that the "==" operator compares the values of the elements in the lists, while the "is" keyword compares the identity of the list objects in memory. In most cases, when comparing lists for equality, you would use the "==" operator. However, if you specifically want to check if two list objects are the same object in memory, you would use the "is" keyword. |

### in KEYWORD

* The "in" keyword is used to check if a specified value exists in a sequence, such as a list, tuple, string, or dictionary. It returns a boolean value indicating whether the value is present or not.

|  |  |
| --- | --- |
| fruits = ['apple', 'banana', 'orange']  if 'banana' in fruits:  print("banana is present in the list") | message = "Hello, world!"  if 'world' in message:  print("The word 'world' is present in the message") |

## 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) |

### ELSE IN LOOP

* **The `else` keyword can be used in conjunction with a `for` loop to specify a block of code that should be executed after the loop completes, but only if the loop did not encounter a `break` statement.**

|  |  |
| --- | --- |
| numbers = [1, 2, 3, 4, 5]  for num in numbers:  if num > 2:  print("Number found!")  break  else:  print("Number not found!") **# This will not execute**  **OUTPUT - Number found** | numbers = [1, 2, 3, 4, 5]  for num in numbers:  if num == 6:  print("Number found!")  break  else:  print("Number not found!") **# This will execute**  **OUTPUT - Number not found** |

* *The `else` block in a `for` loop is optional and can be used to specify additional code to be executed after a loop finishes normally. It provides a way to differentiate between the loop completing normally and the loop being terminated early with a `break` statement.*

### 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”) |

## SOME USEFUL OPERATORS AND FUNCTIONS

### RANGE

* 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.

|  |  |
| --- | --- |
| SYNTAX  **range(start, stop, step)** | * `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. |

**EXAMPLES**

|  |  |
| --- | --- |
| GENERATING A SEQUENCE FROM 0 TO 4 (**EXCLUSIVE**): | for i in range(5):  print(i) |
| GENERATING A SEQUENCE FROM 2 TO 8 (**EXCLUSIVE**) WITH A STEP OF 2 | for i in range(2, 9, 2):  print(i) |
| CREATING A LIST OF NUMBERS WITHIN A RANGE | my\_list = list(range(1, 6))  print(my\_list) |

### SORT

* The `sort()` function in Python has two optional parameters: `**key` and `reverse`.**
* It's important to note that the `sort()` function modifies the original list in place and does not return a new sorted list.

|  |  |
| --- | --- |
| **`key parameter:**   * The `key` parameter allows us to specify a function that will be used to determine the sorting order. * This function is applied to each element of the list before sorting. * The `key` function should take an element as input and return a value that will be used for sorting. * By default, the `key` parameter is `None`, which means the elements are sorted based on their natural order. | **fruits = ["apple", "banana", "cherry", "date"]**  **fruits.sort(key=len)**  **print(fruits)**  **# Output: ['date', 'apple', 'cherry', 'banana']**   * The `key` parameter is set to the `len` function, which returns the length of each element. * The list is sorted based on the length of the elements. |
| **reverse` parameter:**   * The `reverse` parameter is a boolean value that determines whether the list should be sorted in descending order. * By default, it is set to `False`, which means the list is sorted in ascending order. If you set `reverse` to `True`, the list will be sorted in descending order. | **numbers = [5, 2, 9, 1, 7]**  **numbers.sort(reverse=True)**  **print(numbers)**  **# Output: [9, 7, 5, 2, 1]**  In this example, the `reverse` parameter is set to `True`, so the `numbers` list is sorted in descending order, resulting in `[9, 7, 5, 2, 1]`. |

## 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 | |
| **FUNCTION WITH KEYWORD ARGUMENT** | def print\_name(first\_name, last\_name):  print(f"{first\_name},{last\_name}")  print\_name(last\_name='Doe', first\_name='John') | |
| **INVALID** | | **INVALID** |
| **Note : if the first argument has name then all subsequent argument should have name as well**  def print\_name(first\_name, last\_name):  print(f"{first\_name},{last\_name}")  **print\_name(last\_name='Doe', 'John')** | | def print\_name(first\_name, last\_name):  print(f"{first\_name},{last\_name}")  **print\_name('Doe', first\_name= 'John')**  **But if the first argument don’t have name , subsequent arguments can have names** |

### DEFAULT VALUE OF FUNCTION PARAMETERS

|  |  |
| --- | --- |
| * Default values in function parameters are values that are automatically assigned to the parameters if no argument is provided for them when calling the function. * This allows functions to be called with fewer arguments, as the default values will be used for any parameters that are not explicitly passed. | **def greet(name, greeting="Hello"):**  **print(greeting + ", " + name + "!")**    **# Calling the function with only one argument**  **greet("Alice")**  **# Output: Hello, Alice!**    **# Calling the function with two arguments**  **greet("Bob", "Hi")**  **# Output: Hi, Bob!** |
| INVALID | INVALID |
| **Note : if the first parameter has a default value then then all subsequent parameter should have default vale as well**  def add(x=5,y):  return x + y add(y=2) | def add(x,y=5):  return x + y add(y=2)  *TypeError: add() missing 1 required positional argument: 'x'* |
| **DEFAULT VALUE**   * When the function my\_function is defined, the default value for the parameter param\_1 is set to the value of the variable a, which is 10 at that time. This default value is evaluated only once, at the time of function definition. * Later, the value of the variable a is changed to 20. However, this change does not affect the default value of param\_1, as it has already been assigned when the function was defined. * When my\_function() is called without passing any arguments, the default value of param\_1 (which is 10) is used and printed. Hence, the result will be 10, regardless of the subsequent change in the value of a. | a = 10  def my\_function(param\_1=a):  print(param\_1)  a = 20  my\_function()  In the above example, the result will be 10. |

* **In Python, we can't use a function before it is defined. If we try to do so, Python will raise a `NameError` stating that the function name is not defined.**
* In Python, if we call a function with fewer arguments than the number of parameters in the function definition, Python throws a `**TypeError**` indicating that the function is missing some required positional arguments.

### UNPACKING ARGUMENT

#### \*args

|  |  |  |  |
| --- | --- | --- | --- |
| **args : argument**   * 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.  |  | | --- | | **def multiply(\*args):  print(args) 🡪 (1, 2, 3)[Tuple] multiply(1,2,3**) | | | 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 |
| **DESTRUCTURING**  num, \*args = [1, 2, 3, 4, 5] print(num) 🡪 1 print(args) 🡪 [2, 3, 4, 5]  num, \*middle, last = [1, 2, 3, 4, 5] print(num) 🡪 1 print(middle) 🡪 [2, 3, 4] print(last) 🡪 5  \*args, last = [1, 2, 3, 4, 5] print(args) 🡪 [1, 2, 3, 4] print(last) 🡪5 | **DESTRUCTURING**  def multiply(a,b):  return a\*b  input = [3,4] multiply(\*input)  Passing the \*input then get de-structured in function parameters | |

#### \*\*kwargs

|  |  |  |  |
| --- | --- | --- | --- |
| **kwargs :Keyword argument**   * 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. | | def print\_person\_info(\*\*kwargs):  for key, value in kwargs.items():  print(key + ": " + value)    print\_person\_info(name="Alice", age="25", city="New York")  **OR USING DESTRUCTURING**   |  | | --- | | def print\_person\_info(name, age, city):  print(f"{name} is {age} years old and lives in {city}")   person = {"name": "Alice", "age": "25", "city": "New York"} print\_person\_info(\*\*person) | |
| **COMBINING BOTH**  def printData(\*args,\*\*kwargs):  print(args)  print(kwargs)  Function Call : **print(1,2,3,name='Max', age=25)**  **OUTPUT**  (1, 2, 3) 🡪 tuple  {'name': 'Max', 'age': 25} 🡪 Dictionary | * We can mix positional arguments and keyword arguments in a function call**, but the positional arguments must come before the keyword arguments.** | |

## LAMBDA FUNCTIONS, MAP AND FILTERS

* A lambda function is a different type of function which doesn't have a name, and is only used to return values.
* Lambda functions are exclusively used to operate on inputs and return outputs. They are almost never used to perform actions.
* 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 |

### FIRST CLASS FUNCTION

* In Python, a first-class function refers to the concept of treating functions as first-class citizens, which means that
  + FUNCTIONS CAN BE ASSIGNED TO VARIABLES,
  + PASSED AS ARGUMENTS TO OTHER FUNCTIONS,
  + RETURNED AS VALUES FROM OTHER FUNCTIONS.

Essentially, functions are treated as any other data type in the language.

<https://teclado.com/30-days-of-python/python-30-day-16-lambda-expressions/>

|  |  |  |
| --- | --- | --- |
| **ASSIGNING FUNCTIONS TO VARIABLES:**  def greet(name):  print("Hello, " + name + "!")    hello = greet  hello("Alice")  # Output: Hello, Alice! | **PASSING FUNCTIONS AS ARGUMENTS TO OTHER FUNCTIONS**  def apply\_operation(operation, a, b):  return operation(a, b)    def add(a, b):  return a + b    result = apply\_operation(add, 2, 3)  print(result) | |
| **RETURNING FUNCTIONS FROM OTHER FUNCTIONS**  def greet():  def say\_hello(name):  return "Hello, " + name + "!"    return say\_hello    hello = greet()  result = hello("Alice")  print(result)  # Output: Hello, Alice! | **STORING FUNCTIONS IN DATA STRUCTURES**  def square(x):  return x \* x    def cube(x):  return x \* x \* x    **functions = [square, cube]**    result = functions[0](2)  print(result)  # Output: 4 | |
| **EXAMPLES** | | |
| **def divide(a, b):**  **if b == 0:**  **return "You can't divide by 0!"**  **else:**  **return a / b**  operations = {  "a": lambda a, b: a + b,  "s": lambda a, b: a + b,  "m": lambda a, b: a \* b,  "d": **divide** }  selected\_option = input("""Please select one of the following options:  a: add s: subtract m: multiply d: divide  What would you like to do? """) operation = operations.get(selected\_option) if operation:  a = int(input("Please enter a value for a: "))  b = int(input("Please enter a value for b: "))  print(operation(a, b)) else:  print("Invalid selection") | | |
| **`Problem - Use the sort method to put the following list in alphabetical order with regards to the students' names:**  students = [  {"name": "Hannah", "grade\_average": 83},  {"name": "Charlie", "grade\_average": 91},  {"name": "Peter", "grade\_average": 85},  {"name": "Rachel", "grade\_average": 79},  {"name": "Lauren", "grade\_average": 92}  ]  **students.sort(key=lambda student: student["name"])**  **students.sort(key=lambda student: student["name"], reverse=True) 🡪 Sort the student in reverse order of the name** | | **OUTPUT**  **[**  **{'name': 'Charlie', 'grade\_average': 91},**  **{'name': 'Hannah', 'grade\_average': 83},**  **{'name': 'Lauren', 'grade\_average': 92},**  **{'name': 'Peter', 'grade\_average': 85},**  **{'name': 'Rachel', 'grade\_average': 79}**  **]** |

### MAP

* The map() function applies a given function to each item of an iterable and returns a new iterator with the results.
* Map will manipulate the existing list(final length of list will remain same)

|  |  |
| --- | --- |
| **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).
* Filter will filter out the existing list

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

#### EXAMPLE

|  |  |
| --- | --- |
| **PRINT EVEN NUMBERS USING FILTER**  def isEven(num):  if num % 2 == 0:  return num for num in filter(isEven, nums):  print(num) | **USING LAMDAS** for num in **filter**(lambda num : num%2 ==0, nums):  print(num) |

## 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(def or lambda).
* **Local variables can only be accessed within the function in which they are defined.**

|  |  |
| --- | --- |
| def greet **message= "This is global variable"**  ():  **message = "This is enclosed variable"**  def hello():  **message = "This is local variable"**  print("Hello: " + message)  hello()  greet() | * Message is local to the hello() function , hence in the scope resolution it will look for the local variable. (as per L🡪E🡪G🡪B Rule) * **OUTPUT : Hello: This is local variable** |

**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.
  + Example

|  |  |
| --- | --- |
| **message= "This is global variable"**  def greet():  **message = "This is enclosed variable"**  def hello():  print("Hello: " + message)  hello()  greet() | * hello() is the nested function , hence in the scope resolution it will look for the variable enclosed within the enclosing outer function * **OUTPUT : Hello: This is enclosed variable** |
| **message= "This is global variable"**  def greet():  def hello():  print("Hello: " + message)  hello()  greet() | * If there is no local variable it will resolve to the global variable (as per L🡪E🡪G🡪B Rule) * OUTPUT : Hello: This is global variable |

**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.

|  |  |
| --- | --- |
| message= "This is global variable"  def greet():  global message  message = "This is enclosed variable"  greet()  print(message) | * This code illustrates how the `**global**` keyword can be used to modify a global variable within a function, affecting its value outside the function scope. * **OUTPUT: This is enclosed variable** |

**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.**

## INTERACTIVITY – USER INPUTS

|  |  |
| --- | --- |
| * The input() function in Python is used to accept user input from the keyboard. * The return type is “string”. * Hence for any mathematical operation we need to do type conversion | 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()) |

## 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 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 |

### \_\_repr\_\_ and \_\_str\_\_ METHODS

In Python, the `\_\_repr\_\_` and `\_\_str\_\_` methods are special methods that can be defined within a class to provide string representations of objects. These methods are automatically called by certain built-in functions and operators when working with objects of the class.

#### `\_\_str\_\_` method

|  |  |
| --- | --- |
| The `\_\_str\_\_` method is used to provide a string representation of an object that is meant to be readable and user-friendly. It should return a string that represents the object in a more human-readable format. This method is often used when printing or displaying objects. | class Point:  def \_\_init\_\_(self, x, y):  self.x = x  self.y = y    def \_\_str\_\_(self):  return f"({self.x}, {self.y})"    point = Point(2, 3)  print(point) 🡪 (2,3) |

#### `\_\_repr\_\_` method:

|  |  |
| --- | --- |
| * The `\_\_repr\_\_` method is used to provide a string representation of an object that is unambiguous and can be used to recreate the object. * It should return a string that represents the object in a way that, if evaluated, would create an identical object. * This method is primarily used for debugging and development purposes. | class Point:  def \_\_init\_\_(self, x, y):  self.x = x  self.y = y    def \_\_repr\_\_(self):  return f"Point({self.x}, {self.y})"    point = Point(2, 3)  **print(repr(point)) 🡪 Point(2, 3)** |

* By defining both `\_\_repr\_\_` and `\_\_str\_\_` methods, you can control how your objects are represented in different contexts, such as debugging or user output. If only one of these methods is defined, the other method may fall back to using the output of the defined method.

### CLASS AND STATIC METHODS

|  |  |
| --- | --- |
| class Circle:  **pi = 3.14**  def \_\_init\_\_(self, radius):  self.radius = radius    **@classmethod**  **def change\_pi(cls, new\_pi):**  **cls.pi = new\_pi**    **@classmethod**  **def from\_diameter(cls, diameter):**  **radius = diameter / 2**  **return cls(radius) 🡨 CALLING THE CONSTRUCTOR**  circle = Circle(5)  print(circle.pi)  Circle.change\_pi(3.14159)  print(circle.pi)    circle2 = Circle.from\_diameter(10)  print(circle2.radius) | **CLASS METHODS**   * Class methods are methods that are bound to the class and not the instance of the class. * They are defined using the **@classmethod** decorator and **take the class itself as the first parameter**, conventionally named **cls**. * Class methods can be used to manipulate class-level variables or perform operations related to the class. * In this example, the change\_pi class method is used to change the value of the class variable pi. The from\_diameter class method is used as an alternative constructor to create a Circle object based on the diameter. * **USE CASE**: Class methods are useful when you want to perform an operation on the entire class or manipulate class-level variables. For example, in a BankAccount class, a class method could be used to calculate the average balance of all accounts. |
| **SCENARIOS WHERE USING @CLASSMETHOD CAN BE BENEFICIAL** | |
| **Alternative constructors**:  Class methods are often used as alternative constructors. If we have multiple ways to create instances of a class with different input formats or initialization logic, we can define class methods to provide these alternative ways of instantiation. | **class Person:**  **def \_\_init\_\_(self, name, age):**  **self.name = name**  **self.age = age**    **@classmethod**  **def from\_birth\_year(cls, name, birth\_year):**  **age = datetime.date.today().year - birth\_year**  **return cls(name, age)**    **person1 = Person("Alice", 25)**  **person2 = Person.from\_birth\_year("Bob", 1990)** |
| * **Accessing class-level variables or methods**: Class methods have access to the class itself, including class-level variables and other class methods. If we need to perform operations that involve the class as a whole, rather than a specific instance, using a class method can be useful. | **class MathUtils:**  **PI = 3.14**    **@classmethod**  **def calculate\_area(cls, radius):**  **return cls.PI \* radius\*\*2**    **area = MathUtils.calculate\_area(5)** |
| * **Factory methods**: Class methods can be used as factory methods to create and return instances of the class. This allows for more flexibility in object creation by encapsulating the creation logic within the class itself. | **class Car:**  **def \_\_init\_\_(self, make, model):**  **self.make = make**  **self.model = model**    **@classmethod**  **def create\_sedan(cls, make, model):**  **return cls(make, model, "sedan")**    **sedan = Car.create\_sedan("Toyota", "Camry")** |
| * **Polymorphism and method overriding**: Class methods can be overridden in subclasses, providing a way to define polymorphic behavior. This allows subclasses to have their own implementation of the class method while still using the same method signature. class Shape: | @classmethod  **def get\_type(cls):**  **return "Shape"**    **class Circle(Shape):**  **@classmethod**  **def get\_type(cls):**  **return "Circle"**    **shape = Shape()**  **circle = Circle()**    **print(shape.get\_type()) # Output: "Shape"**  **print(circle.get\_type()) # Output: "Circle** |
| class MathUtils:  @staticmethod  def add(x, y):  return x + y    result = MathUtils.add(5, 3)  print(result)  Static methods are handy for utility functions or operations that do not rely on instance or class state. For example, in  StringUtils class, a static method could be used to check if a given string is a palindrome. | **STATIC METHODS**   * Static methods are methods that do not operate on instance or class variables. * They are defined using the **@staticmethod** decorator **and do not require the self or cls parameters**. * *Static methods are often used for utility functions or operations that do not rely on any specific instance or class state* * In this example, the add static method is defined in the MathUtils class. It takes two parameters and returns their sum. Since the method is static, it can be called directly on the class without creating an instance. |

### 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. |

### COMPOSITION

### 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();** |

### MULTIPLE INHERITANCE

A diagram of a software company

Description automatically generated

|  |
| --- |
| **user.py**  from save import Savable class User(Savable):  def \_\_init\_\_(self, user\_name, user\_password):  self.user\_name = user\_name  self.user\_password = user\_password |
| **admin.py**  from user import User from save import Savable class Admin(User, Savable):  def \_\_init\_\_(self, user\_name, user\_password, access\_level):  super().\_\_init\_\_(user\_name, user\_password)  self.access\_level = access\_level   def \_\_repr\_\_(self):  return f'<Admin {self.user\_name}, access level {self.access\_level}>'   def to\_dict(self):  return {  'user\_name': self.user\_name,  'user\_password': self.user\_password,  'access\_level': self.access\_level  } |
| **database.py**  class Database:  content = {'users': []}   @classmethod  def insert(cls, user):  cls.content["users"].append(user)   @classmethod  def remove(cls, finder):  cls.content["users"] = [usr for usr in cls.content["users"] if not finder(usr)]   @classmethod  def find(cls, finder):  found\_users = [usr for usr in cls.content["users"] if finder(usr)]  if found\_users:  return found\_users[0]  return None |
| **Save.py**  from database import Database class Savable:  def save(self):  Database.insert(self.to\_dict()) |

#### MORE EXAMPLES

|  |  |
| --- | --- |
| class Grade:  def calculate\_grade(self, average) -> str:  if average >= 90:  return "A+"  elif 80 <= average < 90:  return "A"  elif 70 <= average < 80:  return "B"  elif 60 <= average < 70:  return "C"  elif 50 <= average < 60:  return "D"  else:  return "F"  class Marks:  def calculate\_average(self) -> float:  return sum(self.marks) / len(self.marks)  class Student(Grade, Marks):  def \_\_init\_\_(self, name, marks):  self.name = name  self.marks = marks   def print\_profile(self):  avg = self.calculate\_average()  return f"Name: {self.name}, Average Marks: {avg} , Grade :{self.calculate\_grade(avg)}"  student = Student('Alex', marks=[90, 80, 70, 60, 50]) print(student.print\_profile()) | *""" Define your Salary class and Promotable class so that the Employee class objects may work like this: rolf = Employee(15.0) print(rolf.weekly\_salary()) # --> prints out rolf's weekly salary (15.0 \* 40 = 600.0) rolf.promote(5.0) # rolf's hourly salary (rate) increases by 5.0 (15.0 + 5.0 = 20.0) print(rolf.weekly\_salary()) # --> prints 800.0 (20.0 \* 40 = 800.0) """* class Salary:  # define Salary class and associated methods here  def calculate(self, hours) -> float:  return hours \* self.rate  class Promotable:  # define Promotable class and associated methods here  def promote(self, increased\_rate) -> None:  self.rate = self.rate + increased\_rate  # Do NOT change the code below:  class Employee(Salary, Promotable):  def \_\_init\_\_(self, rate: float):  # rate is hourly salary  self.rate = rate   def weekly\_salary(self) -> float:  # 40 is number of hours worked/w  return self.calculate(40)  rolf = Employee(15.0) print(rolf.weekly\_salary()) # 600.0 rolf.promote(5.0) print(rolf.weekly\_salary()) # 800.0 |

### ABSTRACT CLASSES

* ABCMeta in Python is a metaclass used to define Abstract Base Classes (ABCs). It is part of the abc module, which helps define classes that cannot be instantiated directly but serve as blueprints for other classes.

**Why Use ABCMeta?**

1. **Enforce Implementation**: Ensure that derived classes implement specific methods.
2. **Define Abstract Methods**: Specify methods that must be overridden in subclasses.
3. **Promote Interface Programming**: Create classes focused on what they should do, not how they do it.

|  |  |
| --- | --- |
| * To work with ABCMeta, we use the ABC class from the abc module as a base class.   **KEY POINTS:**   * ABCMeta is the metaclass behind the scenes for any class inheriting from ABC. * **@abstractmethod**: Decorator to mark methods that must be implemented by subclasses. * We cannot create an instance of a class with unimplemented abstract methods.   **CHECKING IF A CLASS IS A SUBCLASS OF AN ABC**  print(issubclass(Dog, Animal))  # Output: True  print(isinstance(dog, Animal)) # Output: True | **from abc import ABC, abstractmethod**  class Animal(**ABC**):      @abstractmethod      def sound(self):          """This method must be implemented in the subclass."""          pass      @abstractmethod      def move(self):          """This method must be implemented in the subclass."""          pass  # Attempting to instantiate an abstract class will raise an error  # animal = Animal()  # TypeError: Can't instantiate abstract class  class Dog(Animal):      def sound(self):          return "Bark"        def move(self):          return "Run"  # Now you can instantiate Dog  dog = Dog()  print(dog.sound())  # Output: Bark  print(dog.move())   # Output: Run |

---

## TYPE HINTING

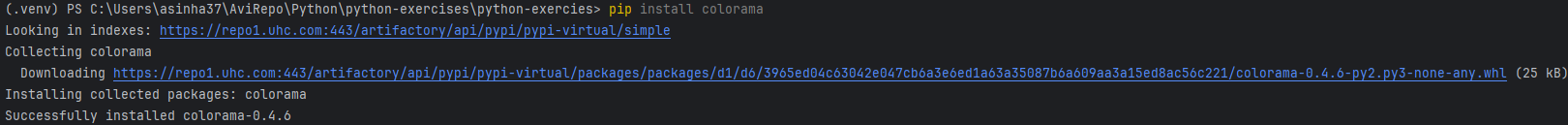
## MODULES AND PACKAGES

### 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() |

### RAISING ERRORS

* 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

|  |  |
| --- | --- |
| **EXAMPLE -1**  age = -1  if age < 0:  raise ValueError("Age cannot be negative") |  |

### CUSTOM ERRORS

#### STEPS TO CREATE CUSTOM EXCEPTIONS

|  |  |
| --- | --- |
| **Step 1: Define a class for custom exception**:   * Create a new class that inherits from the base `Exception` class or any of its subclasses. * This class will serve as custom exception. | class CustomException(Exception):  pass |
| **CUSTOMIZE THE EXCEPTION**   * We can add additional attributes, methods, or customize the behavior of the custom exception by adding them to the class. | class CustomException(Exception):  def \_\_init\_\_(self, message):  self.message = message  super().\_\_init\_\_(self.message)   * In this example, we add an `\_\_init\_\_()` method to the custom exception class to initialize the exception with a custom error message. * The `super().\_\_init\_\_(self.message)` line ensures that the base class `Exception` is also initialized with the custom error message. |
| **RAISE THE CUSTOM EXCEPTION**   * When we encounter a situation that warrants raising the custom exception, use the `**raise**` statement followed by an instance of custom exception class | raise CustomException("This is a custom exception") |
| **Handle the custom exception**   * Surround the code that may raise the custom exception with a `try-except` block to catch and handle the exception | try:  # Code that may raise the custom exception  except CustomException as e:  # Handle the custom exception |

#### EXAMPLES

|  |  |
| --- | --- |
| class InvalidInputException(Exception):  def \_\_init\_\_(self, value):  self.value = value  super().\_\_init\_\_(f"Invalid input: {value} is not a valid value")    def process\_input(value):  if value < 0 or value > 100:  raise InvalidInputException(value)  else:  print(f"Processing input: {value}")    try:  input\_value = 150  process\_input(input\_value)  except InvalidInputException as e:  print(e.value) | * In this example, we define a custom exception class `**InvalidInputException**` that inherits from the base `Exception` class. The `\_\_init\_\_()` method is overridden to initialize the exception with the `value` attribute and a custom error message. * The `process\_input()` function takes a `value` as a parameter and checks if it is a valid input (between 0 and 100). If it is not valid, the function raises an `InvalidInputException` with the value as the argument. Otherwise, it processes the input. * In the `try-except` block, we call the `process\_input()` function with an input value of 150. Since 150 is not a valid value, an `InvalidInputException` is raised. The exception is caught in the `except` block, and the value passed to the exception is printed. |
| class UncountableError(**ValueError**):  def \_\_init\_\_(self**,n**):  super().\_\_init\_\_(f"Invalid value for {n}, WRONG\_VALUE. {n} must be greater than 0.")   def count\_from\_zero\_to\_n(n):  if n < 1:  **raise UncountableError(n)**  for x in range(0, n + 1):  print(x)  count\_from\_zero\_to\_n(-1) | |

## 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() |

## FILES

### 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

### WORKING WITH SPREADSHEETS(CSV)

#### READING CSVs

* There are multiple ways to read / write in a CSV file . In the below example – we will be using built it CSV module in python

|  |
| --- |
| import csv  **# Step 1 - Open the file**  data = open('C:\\Users\\asinha37\\AviRepo\\Python\\python-exercises\\python-exercies\\resources\\example.csv',  encoding='utf-8')  **example.csv**  **# Step 2 - Call csv.reader on data**  data\_csv = csv.reader(data)  **# Step 3 - Reformat it to python object i.e list of list**  data\_lines = list(data\_csv)  for data\_line in data\_lines[:5]:  print(data\_line)  **data.close()**  **OUTPUT**  ['id', 'first\_name', 'last\_name', 'email', 'gender', 'ip\_address', 'city']  ['1', 'Joseph', 'Zaniolini', 'jzaniolini0@simplemachines.org', 'Male', '163.168.68.132', 'Pedro Leopoldo']  ['2', 'Freida', 'Drillingcourt', 'fdrillingcourt1@umich.edu', 'Female', '97.212.102.79', 'Buri']  ['3', 'Nanni', 'Herity', 'nherity2@statcounter.com', 'Female', '145.151.178.98', 'Claver']  ['4', 'Orazio', 'Frayling', 'ofrayling3@economist.com', 'Male', '25.199.143.143', 'Kungur'] |

* Note : The **data\_csv** will be the **list of list**

|  |  |
| --- | --- |
| **PRINT TO 10 EMAILS FROM CSV**   * **Started the iteration from 2nd item as the 1st item is column heading** | for data\_line in data\_lines[1:11]:  print(data\_line[3]) |
| **PRINT LIST OF FULL NAME TO CSV** | for data\_line in data\_lines[1:]:  print("{first\_name},{last\_name}".format(first\_name=data\_line[1], last\_name=data\_line[2])) |

#### WRITING CSVs

|  |  |
| --- | --- |
| **WRITING TO FILE for ANOTHER FILE (example.csv)**  list\_of\_full\_name=[]; **for data\_line in data\_lines[1:]:**  list\_of\_full\_name.append(data\_line[1]+" "+ data\_line[2])  **# Writing to file in csv format** file\_to\_output = open('C:\\Users\\asinha37\\AviRepo\\Python\\python-exercises\\python-exercies\\resources\\full\_name.csv', mode="w", newline= '') **csv\_writer = csv.writer(file\_to\_output, delimiter=',') csv\_writer.writerow(['Full Name'])** for full\_name in list\_of\_full\_name:  csv\_writer.writerow([full\_name])  file\_to\_output.close() | |
| * The `csv\_writer.writerow()` function is used to write a single row to a CSV file using the `**csv**` module. * **It takes a list or tuple as an argument**, where each element represents a column value in the row. | **import csv**  **# Open the CSV file in write mode**  **with open('example.csv', 'w', newline='') as file:**  **csv\_writer = csv.writer(file)**    **# Write a single row to the CSV file**  **csv\_writer.writerow(['a', 'b', 'c'])**   * In the above example, the `csv\_writer.writerow(['a', 'b', 'c'])` statement writes a single row with three columns ('a', 'b', 'c') to the CSV file. |
| * `**csv\_writer.writerows()**` function is used to write multiple rows to a CSV file. * It takes an iterable of rows, where each row is represented as a list or tuple. | **import csv**  **# Open the CSV file in write mode**  **with open('example.csv', 'w', newline='') as file:**  **csv\_writer = csv.writer(file)**    **# Write multiple rows to the CSV file**  **csv\_writer.writerows([['a', 'b', 'c'], ['x', 'y', 'z']])**   * In the above example, the **csv\_writer.writerows([[‘a,’b’,’c’],[‘x’,’y’,’z’]])** statement writes two rows to the CSV file. The first row contains ('a', 'b', 'c') and the second row contains ('x', 'y', 'z'). |

### WORKING WITH PDFs

* We will be using PyPDF2 free open source library to extract and read files

|  |  |
| --- | --- |
| INSTALL PyPDF2 Library | **pip install PyPDF2** |

* Below code demonstrates how to read and extract text from a PDF file using the `**PyPDF2**` library.

|  |
| --- |
| **import PyPDF2**  with open('C:\\Users\\asinha37\\AviRepo\\Python\\python-exercises\\python-exercies\\resources\\Working\_Business\_Proposal.pdf', 'rb') as file:  **reader = PyPDF2.PdfReader(file**)  num\_of\_pages = len(reader.pages)  print("Number of Pages=",num\_of\_pages)  page = reader.pages[0] # reading the first page  print(page.**extract\_text**()) # extracting text from the first page |

* Opens a PDF file in binary read mode (`**'rb'**`). This is done using a `with` statement to ensure the file is properly closed after its contents are read.
* Within the `with` block, a `PdfReader` object is created by passing the opened file to `PyPDF2.PdfReader`. This object allows access to the PDF's content.
* The number of pages in the PDF is determined by checking the length of the `pages` attribute of the `PdfReader` object.
* The code then reads the first page of the PDF by accessing the first element of the `pages` list. The text content of this page is extracted using the `**extract\_text()**` method.

#### WRITING TO PDFs

* To write to PDF files using the PyPDF2 library in Python, qw can perform operations such as merging, splitting, or adding content to existing PDFs(we cannot write in middle of the page in the using PyPDF2).

|  |
| --- |
| **ADDING PAGE TO PDF**  import PyPDF2 with open('C:\\Users\\asinha37\\AviRepo\\Python\\python-exercises\\python-exercies\\resources\\Working\_Business\_Proposal.pdf', 'rb') as file:  reader = PyPDF2.PdfReader(file)  page = reader.pages[0] # reading the first page  print(page.extract\_text())  with open('C:\\Users\\asinha37\\AviRepo\\Python\\python-exercises\\python-exercies\\resources\\Working\_Business\_Proposal\_Page1.pdf', 'ab') as pdf\_file:  writer = PyPDF2.PdfWriter()  writer.add\_page(page)  writer.write(pdf\_file) |

## UNIT TEST

### LINTING - PYLINT

* Pylint is a Python static code analysis tool that **checks code for errors, potential bugs, coding style, and other issues**.
* It provides feedback on code quality, helps identify and fix potential issues, and enforces coding standards.
* It can be used as a command-line tool or integrated into various development environments, such as IDEs or text editors.
* Pylint analyzes Python code based on a set of predefined rules and generates reports with suggestions and warnings.

#### PEP8

* PEP 8 is a set of guidelines for writing Python code in a consistent and readable manner.
* PEP stands for Python Enhancement Proposal, and PEP 8 specifically focuses on the style and formatting of Python code.
* PEP 8 provides recommendations on various aspects of code style, including naming conventions, indentation, line length, comments, imports, and more. Following these guidelines helps improve code readability, maintainability, and collaboration among developers.

|  |
| --- |
| Some key recommendations from PEP 8 include:   * Using lowercase with underscores for variable and function names (e.g., my\_variable, my\_function). * Using CamelCase for class names (e.g., MyClass). * Indenting with four spaces per level. * Limiting lines to a maximum of 79 characters. * Using spaces around operators and after commas. * Writing docstrings to document code and provide usage information. * Importing modules on separate lines and avoiding wildcard imports (from module import \*). |

* There are various tools available that can automatically check code against PEP 8 guidelines, such **as pylint and flake8**. These tools can be integrated into development environments or run as command-line tools to provide feedback on code style violations.

##### SETTING UP PYLINT

|  |  |
| --- | --- |
| INSTALL PYLINT | pip install pylint |
| Python File (lint\_test.py)  a= 1 b = 2 print(a) print(B) | **pylint myexample.py -r y** |
| GENERATED REPORT | |
| \*\*\*\*\*\*\*\*\*\*\*\*\* Module lint\_test  lint\_test.py:4:0: C0304: Final newline missing (missing-final-newline)  lint\_test.py:1:0: C0114: Missing module docstring (missing-module-docstring)  lint\_test.py:1:0: C0103: Constant name "a" doesn't conform to UPPER\_CASE naming style (invalid-name)  lint\_test.py:2:0: C0103: Constant name "b" doesn't conform to UPPER\_CASE naming style (invalid-name)  lint\_test.py:4:6: E0602: Undefined variable 'B' (undefined-variable)  Report  ======  4 statements analysed.  Statistics by type  ------------------  +---------+-------+-----------+-----------+------------+---------+  |type |number |old number |difference |%documented |%badname |  +=========+=======+===========+===========+============+=========+  |module |1 |1 |= |0.00 |0.00 |  +---------+-------+-----------+-----------+------------+---------+  |class |0 |NC |NC |0 |0 |  +---------+-------+-----------+-----------+------------+---------+  |method |0 |NC |NC |0 |0 |  +---------+-------+-----------+-----------+------------+---------+  |function |0 |NC |NC |0 |0 |  +---------+-------+-----------+-----------+------------+---------+  6 lines have been analyzed  Raw metrics  -----------  +----------+-------+------+---------+-----------+  |type |number |% |previous |difference |  +==========+=======+======+=========+===========+  |code |5 |83.33 |NC |NC |  +----------+-------+------+---------+-----------+  |docstring |0 |0.00 |NC |NC |  +----------+-------+------+---------+-----------+  |comment |0 |0.00 |NC |NC |  +----------+-------+------+---------+-----------+  |empty |1 |16.67 |NC |NC |  +----------+-------+------+---------+-----------+  Duplication  -----------  +-------------------------+------+---------+-----------+  | |now |previous |difference |  +=========================+======+=========+===========+  |nb duplicated lines |0 |0 |0 |  +-------------------------+------+---------+-----------+  |percent duplicated lines |0.000 |0.000 |= |  +-------------------------+------+---------+-----------+  Messages by category  --------------------  +-----------+-------+---------+-----------+  |type |number |previous |difference |  +===========+=======+=========+===========+  |convention |4 |4 |4 |  +-----------+-------+---------+-----------+  |refactor |0 |0 |0 |  +-----------+-------+---------+-----------+  |warning |0 |0 |0 |  +-----------+-------+---------+-----------+  |error |1 |1 |1 |  +-----------+-------+---------+-----------+  % errors / warnings by module  -----------------------------  +----------+-------+--------+---------+-----------+  |module |error |warning |refactor |convention |  +==========+=======+========+=========+===========+  |lint\_test |100.00 |0.00 |0.00 |100.00 |  +----------+-------+--------+---------+-----------+  Messages  --------  +-------------------------+------------+  |message id |occurrences |  +=========================+============+  |invalid-name |2 |  +-------------------------+------------+  |undefined-variable |1 |  +-------------------------+------------+  |missing-module-docstring |1 |  +-------------------------+------------+  |missing-final-newline |1 |  +-------------------------+------------+  ------------------------------------------------------------------  Your code has been rated at 0.00/10 (previous run: 0.00/10, +0.00) | |

### TESTING LIBRARY

* Unittest is a built-in Python module that **provides a framework for writing and running unit tests**. Unit testing is a software testing method where individual components, such as functions, classes, or methods, are tested to ensure they work correctly in isolation.
* The unittest module provides a set of classes and methods that help us define test cases and test suites. Test cases are created by subclassing the `**unittest.TestCase**` class and defining test methods within it. Each test method should test a specific aspect of the code being tested.
* The unittest module provides various assertion methods, such as `**assertEqual`, `assertTrue`, `assertFalse**`, etc., to check the expected behavior of the code under test. These assertions help determine if the actual output matches the expected output.
* Test suites can be created to group related test cases together. The `**unittest.TestSuite**` class allows us to combine multiple test cases or test suites into a single suite.
* To run unit tests, we can use the `**unittest.main()**` function or run the tests using test runners provided by various testing frameworks and development environments.