Contents

[PYTHON 2](#_Toc184470111)

[RUNNING PYTHON FILE FROM COMMAND LINE 2](#_Toc184470112)

[PYTHON DATA TYPES 2](#_Toc184470113)

[TYPE CONVERSION 2](#_Toc184470114)

[TYPE CHECKING 3](#_Toc184470115)

[NUMBER MANIPULATION 3](#_Toc184470116)

[DATATYPES 3](#_Toc184470117)

[STRINGS 3](#_Toc184470118)

[LISTS 5](#_Toc184470119)

[DICTIONARIES 5](#_Toc184470120)

[TUPLES 6](#_Toc184470121)

[SETS 7](#_Toc184470122)

[BOOLEANS 7](#_Toc184470123)

[STATEMENTS 8](#_Toc184470124)

[IF-ELSE 8](#_Toc184470125)

[NESTED IF-ELSE 8](#_Toc184470126)

[ELIF 8](#_Toc184470127)

[FOR LOOP 9](#_Toc184470128)

[WHILE LOOP 9](#_Toc184470129)

[SOME USEFUL OPERATORS 10](#_Toc184470130)

[RANGE 10](#_Toc184470131)

# PYTHON

## RUNNING PYTHON FILE FROM COMMAND LINE

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

## PYTHON DATA TYPES

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Example(s)** |
| **int** | Integer numbers without decimal points | x = 5 |
| **float** | Floating-point numbers with decimal points | y = 3.14 |
| **str** | Sequence of characters enclosed in single or double quotes | name = "John" |
| **bool** | Represents the truth values True or False | is\_valid = True |
| **list** | Ordered collection of items enclosed in square brackets | numbers = [1, 2, 3, 4] |
| **tuple** | Immutable ordered collection of items enclosed in parentheses | coordinates = (10, 20) |
| **dict** | 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'} |

## TYPE CONVERSION

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

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

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

## TYPE CHECKING

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

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

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

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

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

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

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

## NUMBER MANIPULATION

|  |  |
| --- | --- |
| int(8/3) | OUTPUT: 2  Trims the decimal points |
| round(8/3,2) | OUTPUT: 2.67  Round it to 2 decimal places |
| 8//3 | It is same as int(8/3) |

## DATATYPES

### STRINGS

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

#### STRINGS ARE IMMUTABLE

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

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

#### STRINGS OPERATIONS

|  |  |  |
| --- | --- | --- |
| **CREATING A STRING**  my\_string = "Hello, World!"  print(my\_string) # Output: Hello, World! | **ACCESSING CHARACTERS IN A STRING**  my\_string = "Hello, World!"  print(my\_string[0]) # Output: H  print(my\_string[7]) # Output: W | **STRING LENGTH**  my\_string = "Hello, World!"  print(len(my\_string)) # Output: 13 |
| **CONCATENATING STRINGS**  string1 = "Hello"  string2 = "World"  concatenated\_string = string1 + " " + string2  print(concatenated\_string) # Output: Hello World | **STRING SLICING**  my\_string = "Hello, World!"  print(my\_string[7:12]) # Output: World  print(my\_string[:5]) # Output: Hello  print(my\_string[7:]) # Output: World! | **STRING METHODS**  my\_string = "Hello, World!"  print(my\_string.upper()) # Output: HELLO, WORLD!  print(my\_string.lower()) # Output: hello, world!  print(my\_string.split(",")) # Output: ['Hello', ' World!']  print(my\_string.replace("Hello", "Hi")) # Output: Hi, World! |

#### STRING FORMATTING

##### FORMAT METHOD

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

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

##### FSTRING

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

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

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

### LISTS

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

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

#### LIST COMPREHENSION

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

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

EXAMPLES

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

### DICTIONARIES

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

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

### TUPLES

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

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

### SETS

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

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

### BOOLEANS

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

#### LOGICAL OPERATORS – CHAINING COMPARISON OPERATOR

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

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

BOOLEAN FUNCTIONS

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

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

## STATEMENTS

### IF-ELSE

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

### NESTED IF-ELSE

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

### ELIF

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

### 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) |
| **TERMINATING THE LOOP WITH BREAK:** | numbers = [1, 2, 3, 4, 5]  for num in numbers:  if num == 3:  **break**  print(num) |
| **TUPLE UNPACKING** | mylist = [(1,2),(3,4),(5,6)] for value1, value2 in mylist:  print("value 1 ={} and value2= {}".format(value1, value2 ))  **OUTPUT** value 1 =1 and value2= 2  value 1 =3 and value2= 4  value 1 =5 and value2= 6 |

### WHILE LOOP

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

### EXAMPLES

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

## I/O FILES

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

### WITH STATEMENT

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

### CURSORS IN FILE READING

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

#### CURSOR-RELATED METHODS FOR FILE READING

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

**EXAMPLE**

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

#### MODES

## SOME USEFUL OPERATORS

### RANGE

## FUNCTIONS

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

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

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

## LAMBDA EXPRESSIONS, MAP AND FILTERS

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

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

### MAP

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

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

### FILTERS

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

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

## NESTED STATEMENT AND SCOPE

## OOPS IN PYTHON

## CLASSES AND OBJECTS

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

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

### CLASS OBJECT ATTRIBUTES

EXAMPPLE

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

## INHERITANCE

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

## POLYMORPHISM