* Python can distinguish among data types such as integers, floats, strings, and Booleans.
* Integers are whole numbers that can be positive or negative.
* Floats include integers as well as decimal numbers between the integers.
* You can convert integers to floats using typecasting, but you cannot convert a float to an integer.
* You can convert integers and floats to strings.
* You can convert an integer or float value to True (1) or False (0).
* Expressions in Python are a combination of values and operations used to produce a single result.
* Expressions perform mathematical operations such as addition, subtraction, multiplication, and so on.
* We use"//" to round off integer divisions, resulting in float values.
* Python follows the order of operations (BODMASS) to perform operations with multiple expressions.
* Variables store and manipulate data, allowing you to access and modify values throughout your code.
* The assignment operator "=" assigns a value to a variable.
* ":" denotes the value of the variable within the code.
* Assigning another value to the same variable overrides the previous value of that variable.
* You can perform mathematical operations on variables using the same or different variables.
* While performing operations with various variables, modifying a value in one variable will lead to changes in the other variables.
* Python string operations involve manipulating text data using tasks such as indexing, concatenation, slicing, and formatting.
* A string is usually written within double quotes or single quotes, including letters, white space, digits, or special characters.
* A string attaches to another variable and is an ordered sequence of characters.
* Characters in a string identify their index numbers, which can be positive or negative.
* We use strings as a sequence to perform sequence operations.
* You can input a stride value to perform slicing while operating on a string.
* Operations like finding the length of the string, combining, concatenating, and replicating, result in a new string.
* You cannot modify an existing string; they are immutable.
* You can perform escape sequences using " " to change the layout of the string.
* In Python, you perform tasks such as searching, modifying, and formatting text data with its pre-built string methods functions.
* You apply a method to a string to change its value, resulting in another string.
* You can perform actions such as changing the case of characters in a string, replacing items in a string, finding items in a string, and so on using pre-built string methods.

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Code Example** |
|  |  |  |
| Comments | Comments are lines of text that are ignored by the Python interpreter when executing the code<./td> | 1. 1 2. # This is a comment   Copied! |
| Concatenation | Combines (concatenates) strings. | Syntax:   1. 1 2. concatenated\_string = string1 + string2   Copied!  Example:   1. 1 2. result = "Hello" + " John"</td>   Copied! |
| Data Types | - Integer - Float - Boolean - String | Example:   1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. x=7 12. # Integer Value 13. y=12.4 14. # Float Value 15. is\_valid = True 16. # Boolean Value 17. is\_valid = False 18. # Boolean Value 19. F\_Name = "John" 20. # String Value   Copied! |
| Indexing | Accesses character at a specific index. | Example:   1. 1 2. 2 3. my\_string="Hello" 4. char = my\_string[0]   Copied! |
| len() | Returns the length of a string. | Syntax:   1. 1 2. len(string\_name)   Copied!  Example:   1. 1 2. 2 3. my\_string="Hello" 4. length = len(my\_string)   Copied! |
| lower() | Converts string to lowercase. | Example:   1. 1 2. 2 3. my\_string="Hello" 4. uppercase\_text = my\_string.lower()   Copied! |
| print() | Prints the message or variable inside `()`. | Example:   1. 1 2. 2 3. print("Hello, world") 4. print(a+b)   Copied! |
| Python Operators | - Addition (+): Adds two values together. - Subtraction (-): Subtracts one value from another. - Multiplication (\*): Multiplies two values. - Division (/): Divides one value by another, returns a float. - Floor Division (//): Divides one value by another, returns the quotient as an integer. - Modulo (%): Returns the remainder after division. | Example:   1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. x = 9 y = 4 9. result\_add= x + y # Addition 10. result\_sub= x - y # Subtraction 11. result\_mul= x \* y # Multiplication 12. result\_div= x / y # Division 13. result\_fdiv= x // y # Floor Division 14. result\_mod= x % y # Modulo</td>   Copied! |
| replace() | Replaces substrings. | Example:   1. 1 2. 2 3. my\_string="Hello" 4. new\_text = my\_string.replace("Hello", "Hi")   Copied! |
| Slicing | Extracts a portion of the string. | Syntax:   1. 1 2. substring = string\_name[start:end]   Copied!  Example:   1. 1 2. my\_string="Hello" substring = my\_string[0:5]   Copied! |
| split() | Splits string into a list based on a delimiter. | Example:   1. 1 2. 2 3. my\_string="Hello" 4. split\_text = my\_string.split(",")   Copied! |
| strip() | Removes leading/trailing whitespace. | Example:   1. 1 2. 2 3. my\_string="Hello" 4. trimmed = my\_string.strip()   Copied! |
| upper() | Converts string to uppercase. | Example:   1. 1 2. 2 3. my\_string="Hello" 4. uppercase\_text = my\_string.upper()   Copied! |
| Variable Assignment | Assigns a value to a variable. | Syntax:   1. 1 2. variable\_name = value   Copied!  Example:   1. 1 2. 2 3. name="John" # assigning John to variable name 4. x = 5 # assigning 5 to variable x |

| **Term** | **Definition** |
| --- | --- |
| AI | AI (artificial intelligence) is the ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings. |
| Application development | Application development, or app development, is the process of planning, designing, creating, testing, and deploying a software application to perform various business operations. |
| Arithmetic Operations | Arithmetic operations are the basic calculations we make in everyday life like addition, subtraction, multiplication and division. It is also called as algebraic operations or mathematical operations. |
| Array of numbers | Set of numbers or objects that follow a pattern presented as an arrangement of rows and columns to explain multiplication. |
| Assignment operator in Python | Assignment operator is a type of Binary operator that helps in modifying the variable to its left with the use of its value to the right. The symbol used for assignment operator is "=". |
| Asterisk | Symbol "\* " used to perform various operations in Python. |
| Backslash | A backslash is an escape character used in Python strings to indicate that the character immediately following it should be treated in a special way, such as being treated as escaped character or raw string. |
| Boolean | Denoting a system of algebraic notation used to represent logical propositions by means of the binary digits 0 (false) and 1 (true). |
| Colon | A colon is used to represent an indented block. It is also used to fetch data and index ranges or arrays. |
| Concatenate | Link (things) together in a chain or series. |
| Data engineering | Data engineers are responsible for turning raw data into information that an organization can understand and use. Their work involves blending, testing, and optimizing data from numerous sources. |
| Data science | Data Science is an interdisciplinary field that focuses on extracting knowledge from data sets which are typically huge in amount. The field encompasses analysis, preparing data for analysis, and presenting findings to inform high-level decisions in an organization. |
| Data type | Data type refers to the type of value a variable has and what type of mathematical, relational or logical operations can be applied without causing an error. |
| Double quote | Symbol “ “ used to represent strings in Python. |
| Escape sequence | An escape sequence is two or more characters that often begin with an escape character that tell the computer to perform a function or command. |
| Expression | An expression is a combination of operators and operands that is interpreted to produce some other value. |
| Float | Python float () function is used to return a floating-point number from a number or a string representation of a numeric value. |
| Forward slash | Symbol “/“ used to perform various operation sin Python |
| Foundational | Denoting an underlying basis or principle; fundamental. |
| Immutable | Immutable Objects are of in-built datatypes like int, float, bool, string, Unicode, and tuple. In simple words, an immutable object can’t be changed after it is created. |
| Integer | An integer is the number zero (0), a positive natural number (1, 2, 3, and so on) or a negative integer with a minus sign (−1, −2, −3, and so on.) |
| Manipulate | Is the process of modifying a string or creating a new string by making changes to existing strings. |
| Mathematical conventions | A mathematical convention is a fact, name, notation, or usage which is generally agreed upon by mathematicians. |
| Mathematical expressions | Expressions in math are mathematical statements that have a minimum of two terms containing numbers or variables, or both, connected by an operator in between. |
| Mathematical operations | The mathematical “operation” refers to calculating a value using operands and a math operator. |
| Negative indexing | Allows you to access elements of a sequence (such as a list, a string, or a tuple) from the end, using negative numbers as indexes. |
| Operands | The quantity on which an operation is to be done. |
| Operators in Python | Operators are used to perform operations on variables and values. |
| Parentheses | Parentheses is used to call an object. |
| Replicate | To make an exact copy of. |
| Sequence | A sequence is formally defined as a function whose domain is an interval of integers. |
| Single quote | Symbol ‘ ‘ used to represent strings in python. |
| Slicing in Python | Slicing is used to return a portion from defined list. |
| Special characters | A special character is one that is not considered a number or letter. Symbols, accent marks, and punctuation marks are considered special characters. |
| Stride value | Stride is the number of bytes from one row of pixels in memory to the next row of pixels in memory. |
| Strings | In Python, Strings are arrays of bytes representing Unicode characters. |
| Substring | A substring is a sequence of characters that are part of an original string. |
| Type casting | The process of converting one data type to another data type is called Typecasting or Type Coercion or Type Conversion. |
| Types in Python | Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. |
| Variables | Variables are containers for storing data values. |

**List**

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Code Example** |
| append() | The `append()` method is used to add an element to the end of a list. | Syntax:   1. 1 2. list\_name.append(element)   Copied!  Example:   1. 1 2. 2 3. fruits = ["apple", "banana", "orange"] 4. fruits.append("mango") print(fruits)   Copied! |
| copy() | The `copy()` method is used to create a shallow copy of a list. | Example 1:   1. 1 2. 2 3. 3 4. my\_list = [1, 2, 3, 4, 5] 5. new\_list = my\_list.copy() print(new\_list) 6. # Output: [1, 2, 3, 4, 5]   Copied! |
| count() | The `count()` method is used to count the number of occurrences of a specific element in a list in Python. | Example:   1. 1 2. 2 3. 3 4. my\_list = [1, 2, 2, 3, 4, 2, 5, 2] 5. count = my\_list.count(2) print(count) 6. # Output: 4   Copied! |
| Creating a list | A list is a built-in data type that represents an ordered and mutable collection of elements. Lists are enclosed in square brackets [] and elements are separated by commas. | Example:   1. 1 2. fruits = ["apple", "banana", "orange", "mango"]   Copied! |
| del | The `del` statement is used to remove an element from list. `del` statement removes the element at the specified index. | Example:   1. 1 2. 2 3. 3 4. my\_list = [10, 20, 30, 40, 50] 5. del my\_list[2] # Removes the element at index 2 print(my\_list) 6. # Output: [10, 20, 40, 50]   Copied! |
| extend() | The `extend()` method is used to add multiple elements to a list. It takes an iterable (such as another list, tuple, or string) and appends each element of the iterable to the original list. | Syntax:   1. 1 2. list\_name.extend(iterable)   Copied!  Example:   1. 1 2. 2 3. 3 4. 4 5. fruits = ["apple", "banana", "orange"] 6. more\_fruits = ["mango", "grape"] 7. fruits.extend(more\_fruits) 8. print(fruits)   Copied! |
| Indexing | Indexing in a list allows you to access individual elements by their position. In Python, indexing starts from 0 for the first element and goes up to `length\_of\_list - 1`. | Example:   1. 1 2. 2 3. 3 4. 4 5. 5 6. my\_list = [10, 20, 30, 40, 50] 7. print(my\_list[0]) 8. # Output: 10 (accessing the first element) 9. print(my\_list[-1]) 10. # Output: 50 (accessing the last element using negative indexing)   Copied! |
| insert() | The `insert()` method is used to insert an element. | Syntax:   1. 1 2. list\_name.insert(index, element)   Copied!  Example:   1. 1 2. 2 3. 3 4. my\_list = [1, 2, 3, 4, 5] 5. my\_list.insert(2, 6) 6. print(my\_list)   Copied! |
| Modifying a list | You can use indexing to modify or assign new values to specific elements in the list. | Example:   1. 1 2. 2 3. 3 4. 4 5. my\_list = [10, 20, 30, 40, 50] 6. my\_list[1] = 25 # Modifying the second element 7. print(my\_list) 8. # Output: [10, 25, 30, 40, 50]   Copied! |
| pop() | `pop()` method is another way to remove an element from a list in Python. It removes and returns the element at the specified index. If you don't provide an index to the `pop()` method, it will remove and return the last element of the list by default | Example 1:   1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. my\_list = [10, 20, 30, 40, 50] 9. removed\_element = my\_list.pop(2) # Removes and returns the element at index 2 10. print(removed\_element) 11. # Output: 30 12. print(my\_list) 13. # Output: [10, 20, 40, 50]   Copied!  Example 2:   1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. my\_list = [10, 20, 30, 40, 50] 9. removed\_element = my\_list.pop() # Removes and returns the last element 10. print(removed\_element) 11. # Output: 50 12. print(my\_list) 13. # Output: [10, 20, 30, 40]   Copied! |
| remove() | To remove an element from a list. The `remove()` method removes the first occurrence of the specified value. | Example:   1. 1 2. 2 3. 3 4. 4 5. my\_list = [10, 20, 30, 40, 50] 6. my\_list.remove(30) # Removes the element 30 7. print(my\_list) 8. # Output: [10, 20, 40, 50]   Copied! |
| reverse() | The `reverse()` method is used to reverse the order of elements in a list | Example 1:   1. 1 2. 2 3. 3 4. my\_list = [1, 2, 3, 4, 5] 5. my\_list.reverse() print(my\_list) 6. # Output: [5, 4, 3, 2, 1]   Copied! |
| Slicing | You can use slicing to access a range of elements from a list. | Syntax:   1. 1 2. list\_name[start:end:step]   Copied!  Example:   1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. my\_list = [1, 2, 3, 4, 5] 14. print(my\_list[1:4]) 15. # Output: [2, 3, 4] (elements from index 1 to 3) 16. print(my\_list[:3]) 17. # Output: [1, 2, 3] (elements from the beginning up to index 2) 18. print(my\_list[2:]) 19. # Output: [3, 4, 5] (elements from index 2 to the end) 20. print(my\_list[::2]) 21. # Output: [1, 3, 5] (every second element)   Copied! |
| sort() | The `sort()` method is used to sort the elements of a list in ascending order. If you want to sort the list in descending order, you can pass the `reverse=True` argument to the `sort()` method. | Example 1:   1. 1 2. 2 3. 3 4. 4 5. my\_list = [5, 2, 8, 1, 9] 6. my\_list.sort() 7. print(my\_list) 8. # Output: [1, 2, 5, 8, 9]   Copied!  Example 2:   1. 1 2. 2 3. 3 4. 4 5. my\_list = [5, 2, 8, 1, 9] 6. my\_list.sort(reverse=True) 7. print(my\_list) 8. # Output: [9, 8, 5, 2, 1]   Copied! |

**Dictionary**

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Code Example** |
| Accessing Values | You can access the values in a dictionary using their corresponding `keys`. | Syntax:   1. 1 2. Value = dict\_name["key\_name"]   Copied!  Example:   1. 1 2. 2 3. name = person["name"] 4. age = person["age"]   Copied! |
| Add or modify | Inserts a new key-value pair into the dictionary. If the key already exists, the value will be updated; otherwise, a new entry is created. | Syntax:   1. 1 2. dict\_name[key] = value   Copied!  Example:   1. 1 2. 2 3. person["Country"] = "USA" # A new entry will be created. 4. person["city"] = "Chicago" # Update the existing value for the same key   Copied! |
| clear() | The `clear()` method empties the dictionary, removing all key-value pairs within it. After this operation, the dictionary is still accessible and can be used further. | Syntax:   1. 1 2. dict\_name.clear()   Copied!  Example:   1. 1 2. grades.clear()   Copied! |
| copy() | Creates a shallow copy of the dictionary. The new dictionary contains the same key-value pairs as the original, but they remain distinct objects in memory. | Syntax:   1. 1 2. new\_dict = dict\_name.copy()   Copied!  Example:   1. 1 2. 2 3. new\_person = person.copy() 4. new\_person = dict(person) # another way to create a copy of dictionary   Copied! |
| Creating a Dictionary | A dictionary is a built-in data type that represents a collection of key-value pairs. Dictionaries are enclosed in curly braces `{}`. | Example:   1. 1 2. 2 3. dict\_name = {} #Creates an empty dictionary 4. person = { "name": "John", "age": 30, "city": "New York"}   Copied! |
| del | Removes the specified key-value pair from the dictionary. Raises a `KeyError` if the key does not exist. | Syntax:   1. 1 2. del dict\_name[key]   Copied!  Example:   1. 1 2. del person["Country"]   Copied! |
| items() | Retrieves all key-value pairs as tuples and converts them into a list of tuples. Each tuple consists of a key and its corresponding value. | Syntax:   1. 1 2. items\_list = list(dict\_name.items())   Copied!  Example:   1. 1 2. info = list(person.items())   Copied! |
| key existence | You can check for the existence of a key in a dictionary using the `in` keyword | Example:   1. 1 2. 2 3. if "name" in person: 4. print("Name exists in the dictionary.")   Copied! |
| keys() | Retrieves all keys from the dictionary and converts them into a list. Useful for iterating or processing keys using list methods. | Syntax:   1. 1 2. keys\_list = list(dict\_name.keys())   Copied!  Example:   1. 1 2. person\_keys = list(person.keys())   Copied! |
| update() | The `update()` method merges the provided dictionary into the existing dictionary, adding or updating key-value pairs. | Syntax:   1. 1 2. dict\_name.update({key: value})   Copied!  Example:   1. 1 2. person.update({"Profession": "Doctor"})   Copied! |
| values() | Extracts all values from the dictionary and converts them into a list. This list can be used for further processing or analysis. | Syntax:   1. 1 2. values\_list = list(dict\_name.values())   Copied!  Example:   1. 1 2. person\_values = list(person.values())   Copied! |

**Sets**

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Code Example** |
| add() | Elements can be added to a set using the `add()` method. Duplicates are automatically removed, as sets only store unique values. | Syntax:   1. 1 2. set\_name.add(element)   Copied!  Example:   1. 1 2. fruits.add("mango")   Copied! |
| clear() | The `clear()` method removes all elements from the set, resulting in an empty set. It updates the set in-place. | Syntax:   1. 1 2. set\_name.clear()   Copied!  Example:   1. 1 2. fruits.clear()   Copied! |
| copy() | The `copy()` method creates a shallow copy of the set. Any modifications to the copy won't affect the original set. | Syntax:   1. 1 2. new\_set = set\_name.copy()   Copied!  Example:   1. 1 2. new\_fruits = fruits.copy()   Copied! |
| Defining Sets | A set is an unordered collection of unique elements. Sets are enclosed in curly braces `{}`. They are useful for storing distinct values and performing set operations. | Example:   1. 1 2. 2 3. empty\_set = set() #Creating an Empty Set 4. fruits = {"apple", "banana", "orange"}   Copied! |
| discard() | Use the `discard()` method to remove a specific element from the set. Ignores if the element is not found. | Syntax:   1. 1 2. set\_name.discard(element)   Copied!  Example:   1. 1 2. fruits.discard("apple")   Copied! |
| issubset() | The `issubset()` method checks if the current set is a subset of another set. It returns True if all elements of the current set are present in the other set, otherwise False. | Syntax:   1. 1 2. is\_subset = set1.issubset(set2)   Copied!  Example:   1. 1 2. is\_subset = fruits.issubset(colors)   Copied! |
| issuperset() | The `issuperset()` method checks if the current set is a superset of another set. It returns True if all elements of the other set are present in the current set, otherwise False. | Syntax:   1. 1 2. is\_superset = set1.issuperset(set2)   Copied!  Example:   1. 1 2. is\_superset = colors.issuperset(fruits)   Copied! |
| pop() | The `pop()` method removes and returns an arbitrary element from the set. It raises a `KeyError` if the set is empty. Use this method to remove elements when the order doesn't matter. | Syntax:   1. 1 2. removed\_element = set\_name.pop()   Copied!  Example:   1. 1 2. removed\_fruit = fruits.pop()   Copied! |
| remove() | Use the `remove()` method to remove a specific element from the set. Raises a `KeyError` if the element is not found. | Syntax:   1. 1 2. set\_name.remove(element)   Copied!  Example:   1. 1 2. fruits.remove("banana")   Copied! |
| Set Operations | Perform various operations on sets: `union`, `intersection`, `difference`, `symmetric difference`. | Syntax:   1. 1 2. 2 3. 3 4. 4 5. union\_set = set1.union(set2) 6. intersection\_set = set1.intersection(set2) 7. difference\_set = set1.difference(set2) 8. sym\_diff\_set = set1.symmetric\_difference(set2)   Copied!  Example:   1. 1 2. 2 3. 3 4. 4 5. combined = fruits.union(colors) 6. common = fruits.intersection(colors) 7. unique\_to\_fruits = fruits.difference(colors) 8. sym\_diff = fruits.symmetric\_difference(colors)   Copied! |
| update() | The `update()` method adds elements from another iterable into the set. It maintains the uniqueness of elements. | Syntax:   1. 1 2. set\_name.update(iterable)   Copied!  Example:   1. 1 2. fruits.update(["kiwi", "grape"] |

Format strings are a way to inject variables into a string in Python. They are used to format strings and produce more human-readable outputs. There are several ways to format strings in Python:

**String interpolation (f-strings)**

Introduced in Python 3.6, f-strings are a new way to format strings in Python. They are prefixed with 'f' and use curly braces {} to enclose the variables that will be formatted. For example:

1. 1
2. 2
3. 3
4. name = "John"
5. age = 30
6. print(f"My name is {name} and I am {age} years old.")

Copied!

This will output:

1. 1
2. My name is John and I am 30 years old.

Copied!

**str.format()**

This is another way to format strings in Python. It uses curly braces {} as placeholders for variables which are passed as arguments in the format() method. For example:

1. 1
2. 2
3. 3
4. name = "John"
5. age = 50
6. print("My name is {} and I am {} years old.".format(name, age))

Copied!

This will output:

1. 1
2. My name is John and I am 50 years old.

Copied!

**% Operator**

This is one of the oldest ways to format strings in Python. It uses the % operator to replace variables in the string. For example:

1. 1
2. 2
3. 3
4. name = "Johnathan"
5. age = 30
6. print("My name is %s and I am %d years old." % (name, age))

Copied!

This will output:

1. 1
2. My name is Johnathan and I am 30 years old.

Copied!

Each of these methods has its own advantages and use cases. However, f-strings are generally considered the most modern and preferred way to format strings in Python due to their readability and performance.

**Additional capabilities**

F-strings are also able to evaluate expressions inside the curly braces, which can be very handy. For example:

1. 1
2. 2
3. 3
4. x = 10
5. y = 20
6. print(f"The sum of x and y is {x+y}.")

Copied!

This will output:

1. 1
2. The sum of x and y is 30.

Copied!

**Raw String (r’’)**

In Python, raw strings are a powerful tool for handling textual data, especially when dealing with escape characters. By prefixing a string literal with the letter ‘r’, Python treats the string as raw, meaning it interprets backslashes as literal characters rather than escape sequences.

Consider the following examples of regular string and raw string:

**Regular string:**

1. 1
2. 2
3. regular\_string = "C:\new\_folder\file.txt"
4. print("Regular String:", regular\_string)

Copied!

This will output:

1. 1
2. 2
3. Regular String: C:
4. ew\_folderile.txt

Copied!

In the regular string regular\_string variable, the backslashes (\n) are interpreted as escape sequences. Therefore, \n represents a newline character, which would lead to an incorrect file path representation.

**Raw string:**

1. 1
2. 2
3. raw\_string = r"C:\new\_folder\file.txt"
4. print("Raw String:", raw\_string)

Copied!

This will output:

1. 1
2. Raw String: C:\new\_folder\file.txt

Copied!

However, in the raw string raw\_string, the backslashes are treated as literal characters. This means that \n is not interpreted as a newline character, but rather as two separate characters, ‘’ and ‘n’. Consequently, the file path is represented exactly as it appears.