**Course: Python for Data Science IBM**

**Course Content**

This course is divided into five modules. You should set a goal to complete at least one module per week.

**Module 1:** Python Basics

* About the Course
* Types
* Expressions and Variables
* String Operations

**Module 2:** Python Data Structures

* Lists and Tuples
* Dictionaries
* Sets

**Module 3:** Python Programming Fundamentals

* Conditions and Branching
* Loops
* Functions
* Exception Handling
* Objects and Classes
* Practice with Python Programming Fundamentals

**Module 4:** Working with Data in Python

* Reading and Writing Files with Open
* Pandas
* Numpy in Python

**Module 5:** APIs and Data Collection

* Simple APIs
* REST APIs, Web Scraping, and Working with Files
* Final Exam

**Summary Week1:**

**Module 1 Summary: Python Basics**

* 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.

**Reading: 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. name = "John"
2. age = 30
3. print(f"My name is {name} and I am {age} years old.")

Copied!

This will output:

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

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

This will output:

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

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

This will output:

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

Copied!

This will output:

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

Copied!

This will output:

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

**Raw string:**

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

Copied!

This will output:

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

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.

**Module 1 Cheat Sheet: Python Basics**

|  |  |  |
| --- | --- | --- |
| **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 |

**Python Data Structures Cheat Sheet**

**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"] |

**odule 2 Summary: Python Data Structures**

Congratulations! You have completed this module. At this point, you know that:

* In Python, we often use tuples to group related data together.Tuples refer to ordered and immutable collections of elements.
* Tuples are usually written as comma-separated elements in parentheses “()".
* You can include strings, integers, and floats in tuples and access them using both positive and negative indices.
* You can perform operations such as combining, concatenating, and slicing on tuples.
* Tuples are immutable, so you need to create a new tuple to manipulate it.
* Tuples, termed nesting, can include other tuples of complex data types.
* You can access elements in a nested tuple through indexing.
* Lists in Python contain ordered collections of items that can hold elements of different types and are mutable, allowing for versatile data storage and manipulation.
* A list is an ordered sequence, represented with square brackets "[]".
* Lists possess mutability, rendering them akin to tuples.
* A list can contain strings, integers, and floats; you can nest lists within it.
* You can access each element in a list using both positive and negative indexing.
* Concatenating or appending a list will result in the modification of the same list.
* You can perform operations such as adding, deleting, splitting, and so forth on a list.
* You can separate elements in a list using delimiters.
* Aliasing occurs when multiple names refer to the same object.
* You can also clone a list to create another list.
* Dictionaries in Python are key-value pairs that provide a flexible way to store and retrieve data based on unique keys.
* Dictionaries consist of keys and values, both composed of string elements.
* You denote dictionaries using curly brackets.
* The keys necessitate immutability and uniqueness.
* The values may be either immutable or mutable, and they allow duplicates.
* You separate each key-value pair with a comma, and you can use color highlighting to make the key more visible.
* You can assign dictionaries to a variable.
* You use the key as an argument to retrieve the corresponding value.
* You can make additions and deletions to dictionaries.
* You can perform an operation on a dictionary to check the key, which results in a true or false output.
* You can apply methods to obtain a list of keys and values in a dictionary.
* Sets in Python are collections of unique elements, useful for tasks such as removing duplicates and performing set operations like union and intersection. Sets lack order.
* Curly brackets "{}" are helpful for defining elements of a set.
* Sets do not contain duplicate items.
* A list passed through the set function generates a set containing unique elements.
* You use “Set Operations” to perform actions such as adding, removing, and verifying elements in a set.
* You can combine sets using the ampersand "&" operator to obtain the common elements from both sets.
* You can use the Union function to combine two sets, including both the common and unique elements from both sets.
* The sub-set method is used to determine if two or more sets are subsets.

**Cheat Sheet: Python Data Structures Part-2**

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Code Example** |
| 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! |
| 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! |
| 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! |
| 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! |
| 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! |
| 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! |
| 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! |
| 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! |
| 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! |
| 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! |

**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()</td>   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 4. Set 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:  is\_superset = set1.issuperset(set2)  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"]) |

Condition & branching

**Conditions and Branching**

Estimated time needed: 10 minutes

**Objective:**

In this reading, you'll learn about:

1. Comparison operators
2. Branching
3. Logical operators

**1. Comparison operations**

Comparison operations are essential in programming. They help compare values and make decisions based on the results.

**Equality operator**

The equality operator == checks if two values are equal. For example, in Python:

1. 1
2. 2
3. 3
4. age = 25
5. if age == 25:
6. print("You are 25 years old.")

Copied!

Here, the code checks if the variable age is equal to 25 and prints a message accordingly.

**Inequality operator**

The inequality operator != checks if two values are not equal:

1. 1
2. 2
3. if age != 30:
4. print("You are not 30 years old.")

Copied!

Here, the code checks if the variable age is not equal to 30 and prints a message accordingly.

**Greater than and less than**

You can also compare if one value is greater than another.

1. 1
2. 2
3. if age>= 20:
4. Print("Yes, the Age is greater than 20")

Copied!

Here, the code checks if the variable age is greater than or equal to 20 and prints a message accordingly.

**2. Branching**

Branching is like making decisions in your program based on conditions. Think of it as real-life choices.

**The IF statement**

Consider a real-life scenario of entering a bar. If you're above a certain age, you can enter; otherwise, you cannot.

1. 1
2. 2
3. 3
4. 4
5. 5
6. age = 20
7. if age >= 21:
8. print("You can enter the bar.")
9. else:
10. print("Sorry, you cannot enter.")

Copied!

Here, you are using the if statement to make a decision based on the age variable.

**The ELIF Statement**

Sometimes, there are multiple conditions to check. For example, if you're not old enough for the bar, you can go to a movie instead.

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. if age >= 21:
8. print("You can enter the bar.")
9. elif age >= 18:
10. print("You can watch a movie.")
11. else:
12. print("Sorry, you cannot do either.")

Copied!

**Real-life example: Automated Teller Machine (ATM)**

When a user interacts with an ATM, the software in the ATM can use branching to make decisions based on the user's input. For example, if the user selects "Withdraw Cash" the ATM can branch into different denominations of bills to dispense based on the amount requested.

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. user\_choice = "Withdraw Cash"
11. if user\_choice == "Withdraw Cash":
12. amount = input("Enter the amount to withdraw: ")
13. if amount % 10 == 0:
14. dispense\_cash(amount)
15. else:
16. print("Please enter a multiple of 10.")
17. else:
18. print("Thank you for using the ATM.")

Copied!

**3. Logical operators**

Logical operators help combine and manipulate conditions.

**The NOT operator**

**Real-life example: Notification settings**

In a smartphone's notification settings, you can use the NOT operator to control when to send notifications. For example, you might only want to receive notifications when your phone is not in "Do Not Disturb" mode.

The not operator negates a condition.

1. 1
2. 2
3. 3
4. is\_do\_not\_disturb = True
5. if not is\_do\_not\_disturb:
6. send\_notification("New message received")

Copied!

**The AND operator**

**Real-life example: Access control**

In a secure facility, you can use the AND operator to check multiple conditions for access. To open a high-security door, a person might need both a valid ID card and a matching fingerprint.

The AND operator checks if all required conditions are true, like needing both keys to open a safe.

1. 1
2. 2
3. 3
4. 4
5. has\_valid\_id\_card = True
6. has\_matching\_fingerprint = True
7. if has\_valid\_id\_card and has\_matching\_fingerprint:
8. open\_high\_security\_door()

Copied!

**The OR operator**

**Real-life example: Movie night decision**

When planning a movie night with friends, you can use the OR operator to decide on a movie genre. You'll choose a movie if at least one person is interested.

The OR operator checks if at least one condition is true. It's like choosing between different movies to watch.

1. 1
2. 2
3. 3
4. 4
5. 5
6. friend1\_likes\_comedy = True
7. friend2\_likes\_action = False
8. friend3\_likes\_drama = False
9. if friend1\_likes\_comedy or friend2\_likes\_action or friend3\_likes\_drama:
10. choose a movie()

Copied!

**Summary**

In this reading, you delved into the most frequently used operator and the concept of conditional branching, which encompasses the utilization of if statements and if-else statements.