* 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.

**Dictionaries**

|  |  |  |
| --- | --- | --- |
| **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"]) |

Python Data Structures

| **Term** | **Definition** |
| --- | --- |
| Aliasing | Aliasing refers to giving another name to a function or a variable. |
| Ampersand | A character typically "&" standing for the word "and." |
| Compound elements | Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. |
| Delimiter | A delimiter in Python is a character or sequence of characters used to separate or mark the boundaries between elements or fields within a larger data structure, such as a string or a file. |
| Dictionaries | A dictionary in Python is a data structure that stores a collection of key-value pairs, where each key is unique and associated with a specific value. |
| Function | A function is a block of code, defining a set procedure, which is executed only when it is called. |
| 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. |
| Intersection | The intersection of two given sets is the largest set, which contains all the elements that are common to both sets. |
| Keys | The keys () method in Python Dictionary, returns a view object that displays a list of all the keys in the dictionary in order of insertion using Python. |
| Lists | A list is any list of data items, separated by commas, inside square brackets. |
| Logic operations | In Python, logic operations refer to the use of logical operators such as "and," "or," and "not" to perform logical operations on Boolean values (True or False). |
| Mutable | Immutable objects are of in-built datatypes like int, float, bool, string, Unicode, and tuple. A mutable object can be changed after it is created. |
| Nesting | A nested function is simply a function within another function and is sometimes called an "inner function". |
| Ratings in python | Ratings in Python typically refer to a numerical or qualitative measure assigned to something to indicate its quality, performance, or value. |
| Set operations | Set operations in Python refer to mathematical operations performed on sets, which are unordered collections of unique elements. |
| Sets in python | A set is an unordered collection of unique elements. |
| Syntax | The rules that define the structure of the language for python is called its syntax. |
| Tuples | These are used store multiple items in a single variable. |
| Type casting | In python, this is converting one data type to another. |
| Variables | In python, a variable is a symbolic name or identifier used to store and manipulate data. Variables serve as containers for values, and these values can be of various data types, including numbers, strings, lists, and more. |
| Venn diagram | A Venn diagram is a graphical representation that uses overlapping circles to illustrate the relationships and commonalities between sets or groups of items. |
| Versatile data | Versatile data, in a general context, refers to data that can be used in multiple ways, is adaptable to different applications or purposes, and is not restricted to a specific use case. |

**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()