www.hiringhustle.com

@hiringhustle

Python 2025
Interview Kit PDF

HIRING HUSTLE

HiringHustle Python Interview kit-1

Note to Readers

This book is designed to cater to a wide audience, including college students, individuals preparing for interviews, working professionals, and anyone looking to strengthen their Python programming skills. It serves as a comprehensive guide to core and advanced Python topics, presented in an easy-to-understand format.

This is **Part One**, containing **50 chapters**:

1. Introduction to Python
2. Data Types and Variables
3. Basic Operations
4. Conditional Statements
5. Loops
6. Lists
7. Dictionaries
8. Sets
9. Tuples
10. Strings
11. Functions
12. Recursion
13. File I/O
14. Error and Exception Handling
15. Object-Oriented Programming
16. Regular Expressions
17. Iterators and Generators
18. Decorators
19. Comprehensions
20. Multithreading
21. Multiprocessing
22. Modules and Packages
23. Date and Time
24. Linked Lists
25. Filter
26. Heapq
27. Tuple

28. Basic Input and Output
29. Files & Folders I/O
30. os.path
31. Iterables and Iterators
32. Defining Functions with List Arguments
33. Functional Programming in Python
34. Partial Functions
35. Classes
36. Metaclasses
37. String Formatting
38. String Methods
39. Using Loops Within Functions
40. Importing Modules
41. Difference Between Module and Package
42. Math Module
43. Complex Math
44. Collections Module
45. Operator Module
46. JSON Module
47. Sqlite3 Module

More chapters and advanced topics will be added in future editions to further enrich your learning experience.

Stay tuned for updates and enhancements!

Chapter 1: Getting Started with Python Language

This chapter introduces the basics of Python programming. It covers the fundamental concepts that will help you start writing and running Python programs. From setting up the Python environment to understanding the basic

syntax and concepts, this chapter lays the foundation for your Python learning journey.

Section 1.1: Getting Started

To begin programming in Python, you'll need to install Python on your computer. Python is available for multiple operating systems, including Windows, macOS, and Linux. You can download the latest version of Python from the official Python website.

Running Python Code

Once installed, you can run Python using:

- **Python IDE**: IDLE (Integrated Development and Learning Environment) comes pre-installed with Python.
- **Command Line/Terminal**: You can also run Python scripts using the terminal (e.g., `python script.py`).

Python supports both interactive and script-based programming. In interactive mode, you can type and execute Python code directly in the terminal or IDLE.

Section 1.2: Creating Variables and Assigning Values

In Python, you don't need to declare a variable before using it. Variables are created when you first assign a value to them. Python is dynamically typed, meaning you don’t need to specify the type of the variable.

Examples:

```
python
Copy code
# Assigning values to variables
x = 5
name = "John"
price = 12.99
is_active = True

# You can also reassign variables
```

```
x = "Hello"
```

Variables can hold values of various types, and Python will automatically handle the type based on the assigned value.

Section 1.3: Block Indentation

Python uses indentation to define blocks of code. This is crucial as Python doesn't use curly braces (`{}`) to mark the beginning and end of code blocks, unlike many other languages. Proper indentation is necessary for Python code to run correctly.

Example:

```
python
Copy code
def greet(name):
    print(f"Hello, {name}!")  # This line is indented

greet("Alice")  # Function call
```

If you don't maintain correct indentation, Python will throw an `IndentationError`.

Chapter 2: Python Data Types

In this chapter, we will explore the core data types in Python. Understanding data types is essential for working with different kinds of information in Python, from strings and numbers to more complex collections like lists, sets, and dictionaries. Each data type has its own properties and methods, which are used to manipulate data efficiently.

Section 2.1: String Data Type

A **string** is a sequence of characters enclosed in quotes, either single (`'`) or double (`"`). Strings in Python are immutable, meaning once created, their values cannot be changed.

String Operations:

- **Concatenation**: Combine strings using the `+` operator.

```
python
Copy code
greeting = "Hello"
name = "Alice"
message = greeting + " " + name  # "Hello Alice"
```

- **Repetition**: Repeat strings using the `*` operator.

```
python
Copy code
word = "Python"
print(word * 3)  # Output: PythonPythonPython
```

- **String Length**: Use the `len()` function to find the length of a string.

```
python
Copy code
text = "Python"
print(len(text))  # Output: 6
```

- **Accessing Characters**: Strings support indexing and slicing.

```
python
Copy code
word = "Python"
print(word[0])  # Output: P
print(word[1:4])  # Output: yth
```

Methods:

- `str.upper()`: Converts all characters to uppercase.

```
python
Copy code
text = "hello"
print(text.upper())  # Output: HELLO
```

- `str.lower()`: Converts all characters to lowercase.

```
python
Copy code
text = "HELLO"
print(text.lower())  # Output: hello
```

- `str.replace()`: Replaces a substring with another.

```
python
Copy code
text = "I love Python"
print(text.replace("love", "hate"))  # Output: I hate Py
thon
```

Section 2.2: Set Data Type

A **set** is an unordered collection of unique elements. Sets are useful for membership testing and removing duplicates from a collection. Sets are mutable, meaning you can add or remove elements after creation, but they do not allow duplicates.

Creating a Set:

```
python
Copy code
my_set = {1, 2, 3, 4, 5}
```

Set Operations:

- **Adding Elements**: Use `add()` to add an element to a set.

```
python
Copy code
my_set.add(6)
```

- **Removing Elements**: Use `remove()` or `discard()` to remove an element. `remove()` will raise an error if the element is not present, while `discard()` does not.

```
python
Copy code
my_set.remove(3)
my_set.discard(7)  # Does nothing as 7 is not in the set
```

- **Set Union**: Combine two sets using the `|` operator or `union()` method.

```
python
Copy code
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2  # Output: {1, 2, 3, 4, 5}
```

- **Set Intersection**: Get common elements between two sets using the `&` operator or `intersection()` method.

```
python
Copy code
intersection_set = set1 & set2  # Output: {3}
```

Section 2.3: Numbers Data Type

Python has several numeric types, including integers (`int`), floating-point numbers (`float`), and complex numbers (`complex`).

Integer:

```
python
Copy code
x = 5  # Integer
```

Floating-point:

```
python
Copy code
y = 3.14  # Float
```

Complex Numbers:

Complex numbers consist of a real and imaginary part, represented as `a + bj`, where `a` and `b` are numbers, and `j` is the imaginary unit.

```
python
Copy code
z = 3 + 5j  # Complex number
```

Mathematical Operations:

Python supports basic arithmetic operations:

```
python
Copy code
a = 10
b = 3
print(a + b)  # Addition: 13
print(a - b)  # Subtraction: 7
print(a * b)  # Multiplication: 30
```

```
print(a / b)  # Division: 3.3333...
print(a // b)  # Floor Division: 3
print(a % b)  # Modulus: 1
print(a ** b)  # Exponentiation: 1000
```

Section 2.4: List Data Type

A **list** is an ordered collection of items, which can be of different data types. Lists are mutable, so you can change their elements after creation.

Creating a List:

```
python
Copy code
my_list = [1, 2, 3, "Python", 4.5]
```

List Operations:

- **Accessing Elements**: Use indexing to access individual elements.

  ```
  python
  Copy code
  print(my_list[0])  # Output: 1
  ```

- **Slicing**: Extract a sublist using slicing.

  ```
  python
  Copy code
  print(my_list[1:4])  # Output: [2, 3, 'Python']
  ```

- **Appending and Inserting**:

  ```
  python
  Copy code
  ```

```
my_list.append(6)  # Adds 6 at the end
my_list.insert(2, "New Item")  # Inserts at index 2
```

- **Removing Elements**:

```
python
Copy code
my_list.remove("Python")  # Removes the first occurrence
of "Python"
my_list.pop(1)  # Removes element at index 1
```

Section 2.5: Dictionary Data Type

A **dictionary** is an unordered collection of key-value pairs. Keys must be unique and immutable, while values can be of any data type. Dictionaries are mutable, meaning you can change the value of a key.

Creating a Dictionary:

```
python
Copy code
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
```

Accessing Values:

```
python
Copy code
print(my_dict["name"])  # Output: Alice
```

Adding or Updating Items:

```
python
Copy code
```

```
my_dict["age"] = 26  # Update the value
my_dict["country"] = "USA"  # Add a new key-value pair
```

Removing Items:

```
python
Copy code
del my_dict["city"]  # Removes the key "city"
```

Section 2.6: Tuple Data Type

A **tuple** is an ordered collection of items, similar to a list, but tuples are immutable. Once a tuple is created, its values cannot be changed.

Creating a Tuple:

```
python
Copy code
my_tuple = (1, 2, 3, "Python")
```

Accessing Elements:

```
python
Copy code
print(my_tuple[1])  # Output: 2
```

Slicing:

```
python
Copy code
print(my_tuple[1:3])  # Output: (2, 3)
```

Tuples are often used when you want to ensure that the data does not change during execution.

Conclusion

In this chapter, we have covered the basic data types in Python: **strings**, **sets**, **numbers**, **lists**, **dictionaries**, and **tuples**. Understanding these data types will help you manipulate data effectively in Python programs. Each data type has its own strengths, and choosing the right one depends on the problem you're solving.

Chapter 3: Indentation

Indentation is a critical part of writing Python code. Unlike many other programming languages that use curly braces `{}` or keywords to define code blocks, Python uses indentation to structure code. This chapter explains the importance of indentation, how it is parsed by Python, and common errors that can occur due to incorrect indentation.

Section 3.1: Simple Example

In Python, indentation is used to define the scope of loops, functions, classes, conditionals, and other code blocks. The code inside the block is indented, typically by 4 spaces or a tab, but spaces are preferred in most Python style guides.

Example: Basic Python Indentation

```
python
Copy code
def greet():
    print("Hello, World!")

greet()
```

In this example:

- The `print("Hello, World!")` line is indented to indicate that it belongs to the `greet()` function.
- The indentation allows Python to identify that the `print` statement is part of the `greet` function's block.

Another Example: Conditionals

```
python
Copy code
x = 5
if x > 3:
    print("x is greater than 3")
    print("This is inside the if block")
else:
    print("x is less than or equal to 3")
```

In this example:

- The code inside the `if` and `else` blocks is indented.
- Without indentation, Python wouldn't know which statements belong to which block.

Section 3.2: How Indentation is Parsed

Python uses indentation to group statements into blocks, which can then be treated as a unit. When Python executes code, it reads the indentation level to determine which statements belong to which block.

- **Consistent Indentation**: Python requires that the same level of indentation be used for all statements in a block.
- **Tabs vs. Spaces**: Python allows both tabs and spaces for indentation, but it is important to choose one style and use it consistently throughout your code. Mixing tabs and spaces can lead to errors.

How Python Processes Indentation:

- The first line of a block is indented, and all subsequent lines within that block must be indented at the same level.

- Python uses the indentation level to decide whether a line is part of a current block or if it's a new block.

Example: Indentation with Loops

```
python
Copy code
for i in range(3):
    print(i)  # Indented inside the for loop
print("Done")  # Not indented, outside the for loop
```

- The `print(i)` statement is part of the `for` loop because it's indented.
- The `print("Done")` statement is not part of the loop because it's not indented.

Section 3.3: Indentation Errors

Python is very strict about indentation. If your code is not properly indented, Python will raise an `IndentationError` or `TabError`. Here are some common indentation errors:

1. Missing Indentation

```
python
Copy code
def greet():
print("Hello!")  # IndentationError: expected an indented b
lock
```

- The line `print("Hello!")` must be indented to indicate it belongs to the `greet()` function.

2. Inconsistent Indentation (Tabs vs. Spaces)

```
python
Copy code
def greet():
```

```
    print("Hello")  # Indented with spaces
    print("World")   # Indented with a tab
```

- This will cause a `TabError` because Python does not allow mixing spaces and tabs. Use spaces or tabs consistently, and PEP 8 recommends using 4 spaces per indentation level.

3. Unnecessary Indentation

```
python
Copy code
if True:
    print("True")
        print("Indented incorrectly")  # IndentationError:
unexpected indent
```

- The second `print("Indented incorrectly")` line is indented too much and leads to an error. Python expects this to be at the same level as the previous line inside the `if` block.

4. Indentation in the Wrong Scope

```
python
Copy code
if True:
    print("Inside if")
print("Outside if")
    print("Wrong indentation")  # IndentationError: unexpec
ted indent
```

- The second `print("Wrong indentation")` line is incorrectly indented. It should be at the same level as `print("Outside if")`.

Chapter 4: Comments and Documentation

Writing clear and effective comments and documentation is an essential part of writing maintainable code. In this chapter, we'll explore how to add comments to your Python code, how to access and write documentation, and how docstrings help both developers and tools to understand the purpose and behavior of your code.

Section 4.1: Single Line, Inline, and Multiline Comments

1. Single Line Comments

A **single-line comment** starts with the hash symbol (`#`). Everything following the `#` on that line will be treated as a comment.

```
python
Copy code
# This is a single-line comment
x = 10  # This is an inline comment
```

- In the first example, the entire line is a comment.
- In the second example, the comment follows the code on the same line and is known as an **inline comment**.

2. Multiline Comments

Python doesn't have a specific syntax for multiline comments, but you can use consecutive `#` symbols to create comments over multiple lines. Alternatively, you can use triple quotes (`'''` or `"""`) to create a multiline string that acts as a comment, though this is technically a string, not a comment.

```
python
Copy code
# This is a multiline comment
# that continues on the next line
# and goes on until the end.

'''
This is a multiline string.
```

```
It will not be executed, and is often used as a comment blo
ck.
'''
```

- The `#` symbol is the preferred way to write multiline comments for clarity, especially when commenting on sections of code.
- Triple quotes can be used for multiline strings that are not assigned to a variable, and they can also be used for **docstrings** (explained later).

Section 4.2: Programmatically Accessing Docstrings

Docstrings are string literals that appear at the beginning of a function, class, or module and are used to describe its purpose. Python provides a built-in way to access docstrings programmatically using the `help()` function or by directly accessing the `.__doc__` attribute.

Example: Accessing Docstrings with `help()`

```
python
Copy code
def greet():
    """This function prints a greeting message."""
    print("Hello, World!")

help(greet)
```

- The `help()` function displays the docstring of the function, class, or module.
- In this case, it will show the message: `"This function prints a greeting message."`

Accessing Docstrings with `.__doc__`

```
python
Copy code
print(greet.__doc__)  # Output: This function prints a gree
```

```
ting message.
```

- The `.__doc__` attribute is a string containing the docstring of the function. You can use it to retrieve and manipulate documentation programmatically.

Section 4.3: Write Documentation Using Docstrings

Docstrings are used to provide documentation for your code. By placing a string literal inside a function, class, or module, you describe what it does, its parameters, and what it returns (if applicable). Python's documentation tools, such as `help()`, rely on these docstrings.

Docstring Format

The recommended format for docstrings follows this structure:

- A short description of the function, class, or module.
- A description of the parameters (if applicable), including types and purpose.
- A description of the return value (if applicable), including its type.
- Optional: Examples of how to use the function or class.

1. Function Docstrings

Here's an example of a properly documented function:

```
python
Copy code
def add_numbers(a, b):
    """
    Adds two numbers and returns the result.

    Parameters:
    a (int, float): The first number to add.
    b (int, float): The second number to add.

    Returns:
    int, float: The sum of the two numbers.
```

```
    Example:
    >>> add_numbers(2, 3)
    5
    """
    return a + b
```

- The **short description** explains what the function does.
- The **Parameters** section describes each parameter, including its type and purpose.
- The **Returns** section explains what the function returns and its type.
- The **Example** section provides a sample usage of the function.

2. Class Docstrings

Classes can also be documented with docstrings. Here's an example:

```
python
Copy code
class Car:
    """
    A class representing a car.

    Attributes:
    make (str): The make of the car.
    model (str): The model of the car.
    year (int): The year the car was manufactured.

    Methods:
    start_engine(): Starts the car's engine.
    stop_engine(): Stops the car's engine.
    """

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
```

```
    def start_engine(self):
        print(f"{self.make} {self.model}'s engine starte
d.")

    def stop_engine(self):
        print(f"{self.make} {self.model}'s engine stoppe
d.")
```

- The **Attributes** section lists all the attributes of the class with descriptions.
- The **Methods** section lists all the methods and a short description of each.

3. Module Docstrings

You can also document entire Python files (modules) by placing a docstring at the top of the file:

```
python
Copy code
"""
This module provides utility functions for working with str
ings.

Functions:
- capitalize_first_letter(): Capitalizes the first letter o
f a string.
- reverse_string(): Reverses a string.
"""
```

- This docstring provides an overview of what the module does and lists the available functions.

Chapter 5: Date and Time

In this chapter, we will explore how to work with dates and times in Python using the `datetime` module and other useful libraries. Handling dates and times accurately is essential in various applications, and Python provides powerful tools to manage and manipulate date and time data effectively.

Section 5.1: Parsing a String into a Timezone-Aware Datetime Object

Parsing a string into a timezone-aware `datetime` object allows you to handle timestamps correctly across different time zones. You can use the `datetime.strptime()` method to parse a string into a `datetime` object and then make it timezone-aware using the `pytz` library.

Example:

```
python
Copy code
from datetime import datetime
import pytz

# Parse string into datetime
dt_str = "2024-12-29 15:30:00"
dt = datetime.strptime(dt_str, "%Y-%m-%d %H:%M:%S")

# Make it timezone-aware
timezone = pytz.timezone("America/New_York")
dt_aware = timezone.localize(dt)
print(dt_aware)
```

- Here, `strptime()` parses the string into a naive `datetime` object.
- Then, `pytz.timezone().localize()` makes it timezone-aware.

Section 5.2: Constructing Timezone-Aware Datetimes

Creating timezone-aware `datetime` objects from scratch ensures that time information is associated with a specific timezone.

Example:

```
python
Copy code
from datetime import datetime
import pytz
```

```
# Construct a timezone-aware datetime object
timezone = pytz.timezone("Europe/London")
dt = datetime(2024, 12, 29, 15, 30, 0, tzinfo=timezone)
print(dt)
```

- This creates a `datetime` object and associates it with the London timezone.
- The `tzinfo` parameter makes the datetime object timezone-aware.

Section 5.3: Computing Time Differences

You can compute time differences by subtracting two `datetime` objects, which results in a `timedelta` object. This object contains information such as the number of days, seconds, and microseconds between the two datetimes.

Example:

```
python
Copy code
from datetime import datetime

dt1 = datetime(2024, 12, 29, 10, 0, 0)
dt2 = datetime(2024, 12, 29, 15, 30, 0)

time_diff = dt2 - dt1
print("Time Difference:", time_diff)
```

- The result is a `timedelta` object, which in this case will show a difference of 5 hours and 30 minutes.

Section 5.4: Basic Datetime Objects Usage

Datetime objects in Python can be used to represent dates and times. You can create `datetime` objects and access various properties such as year, month, day, hour, minute, second, etc.

Example:

```
python
Copy code
from datetime import datetime

# Current date and time
now = datetime.now()

print("Year:", now.year)
print("Month:", now.month)
print("Day:", now.day)
print("Hour:", now.hour)
print("Minute:", now.minute)
print("Second:", now.second)
```

- `datetime.now()` returns the current local date and time as a `datetime` object.
- You can access individual components like year, month, day, hour, minute, and second.

Section 5.5: Switching Between Time Zones

Switching between time zones is crucial when you need to convert a datetime from one time zone to another. The `pytz` library helps in converting timezone-aware datetime objects.

Example:

```
python
Copy code
from datetime import datetime
import pytz

# Timezone-aware datetime
timezone_utc = pytz.utc
utc_time = datetime.now(timezone_utc)

# Convert UTC to New York time
timezone_ny = pytz.timezone("America/New_York")
```

```
ny_time = utc_time.astimezone(timezone_ny)
print("New York Time:", ny_time)
```

- `astimezone()` converts a timezone-aware `datetime` object to another time zone.

Section 5.6: Simple Date Arithmetic

You can perform basic arithmetic with `datetime` objects, such as adding or subtracting time. You can use `timedelta` objects for this purpose.

Example:

```
python
Copy code
from datetime import datetime, timedelta

# Current date and time
now = datetime.now()

# Add 1 day to the current date
new_date = now + timedelta(days=1)
print("Tomorrow's Date:", new_date)

# Subtract 2 days from the current date
previous_date = now - timedelta(days=2)
print("Two Days Ago:", previous_date)
```

- `timedelta(days=1)` allows you to add or subtract days to a `datetime` object.

Section 5.7: Converting Timestamp to Datetime

You can convert a Unix timestamp (the number of seconds since January 1, 1970) into a `datetime` object using `datetime.fromtimestamp()`.

Example:

```
python
Copy code
from datetime import datetime

# Convert Unix timestamp to datetime
timestamp = 1672531199
dt = datetime.fromtimestamp(timestamp)
print("Converted DateTime:", dt)
```

- `fromtimestamp()` converts a Unix timestamp into a local `datetime` object.

Section 5.8: Subtracting Months from a Date Accurately

Subtracting months from a `datetime` object can be tricky because months have varying lengths. You can use the `dateutil.relativedelta` library for accurate month subtraction.

Example:

```
python
Copy code
from datetime import datetime
from dateutil.relativedelta import relativedelta

dt = datetime(2024, 12, 29)
new_date = dt - relativedelta(months=2)
print("Date After Subtracting 2 Months:", new_date)
```

- `relativedelta` from `dateutil` allows for more complex date manipulations, including accurate month subtraction.

Section 5.9: Parsing an Arbitrary ISO 8601 Timestamp with Minimal Libraries

ISO 8601 timestamps are commonly used to represent dates and times. You can parse ISO 8601 strings into `datetime` objects using Python's built-in `datetime`

module.

Example:

```
python
Copy code
from datetime import datetime

iso_string = "2024-12-29T15:30:00Z"
dt = datetime.fromisoformat(iso_string.replace("Z", "+00:0
0"))
print("Parsed DateTime:", dt)
```

- The `fromisoformat()` method can parse ISO 8601 formatted strings, converting them into `datetime` objects.

Section 5.10: Get an ISO 8601 Timestamp

You can convert a `datetime` object to an ISO 8601 formatted string using `datetime.isoformat()`.

Example:

```
python
Copy code
from datetime import datetime

dt = datetime.now()
iso_string = dt.isoformat()
print("ISO 8601 Timestamp:", iso_string)
```

- The `isoformat()` method converts the `datetime` object to a string in ISO 8601 format.

Section 5.11: Parsing a String with a Short Time Zone Name into a Timezone-Aware Datetime Object

You can parse a datetime string with a short time zone name (e.g., `EST`, `UTC`) into a timezone-aware `datetime` object using the `pytz` library.

Example:

```
python
Copy code
from datetime import datetime
import pytz

dt_str = "2024-12-29 15:30:00 EST"
dt = datetime.strptime(dt_str, "%Y-%m-%d %H:%M:%S %Z")

# Convert to a timezone-aware datetime
timezone = pytz.timezone("US/Eastern")
dt_aware = timezone.localize(dt)
print("Timezone-Aware DateTime:", dt_aware)
```

Section 5.12: Fuzzy Datetime Parsing (Extracting Datetime Out of a Text)

Fuzzy parsing allows extracting date and time from arbitrary text. The `dateutil.parser.parse()` function can be used for this purpose.

Example:

```
python
Copy code
from dateutil import parser

text = "The meeting is scheduled for 2024-12-29 at 3:00 PM"
dt = parser.parse(text)
print("Extracted DateTime:", dt)
```

- `dateutil.parser.parse()` can handle various date formats and extract the date from a string.

Section 5.13: Iterate Over Dates

You can iterate over a range of dates using `datetime` and `timedelta`.

Example:

```
python
Copy code
from datetime import datetime, timedelta

start_date = datetime(2024, 12, 1)
end_date = datetime(2024, 12, 5)

current_date = start_date
while current_date <= end_date:
    print(current_date.strftime("%Y-%m-%d"))
    current_date += timedelta(days=1)
```

- This loop prints each date from `2024-12-01` to `2024-12-05`.

Chapter 6: Date Formatting

In this chapter, we will explore how to format dates and times using Python's `datetime` module. Understanding date formatting is essential for presenting or storing dates in a specific format, parsing date strings, and calculating time differences.

Section 6.1: Time Between Two Date-Times

One of the most common operations with dates is calculating the difference between two `datetime` objects. Python's `datetime` module allows you to subtract two `datetime` objects and return a `timedelta` object, which represents the difference between the two times.

Example:

```
python
Copy code
from datetime import datetime

# Define two datetime objects
start_date = datetime(2024, 12, 1, 10, 0, 0)
end_date = datetime(2024, 12, 29, 15, 30, 0)

# Calculate the time difference
time_difference = end_date - start_date
print("Time Difference:", time_difference)

# Access specific attributes of the timedelta object
print("Days:", time_difference.days)
print("Seconds:", time_difference.seconds)
print("Total seconds:", time_difference.total_seconds())
```

- In this example, the difference between `start_date` and `end_date` is calculated and printed.
- The `timedelta` object provides the difference in days, seconds, and total seconds.

Section 6.2: Outputting Datetime Object to String

To display a `datetime` object in a specific string format, you can use the `strftime()` method, which allows you to format the datetime in a readable way.

Example:

```
python
Copy code
from datetime import datetime

# Current date and time
now = datetime.now()
```

```
# Format datetime as string
formatted_date = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted Date:", formatted_date)

# Other common format codes:
# %Y - Year with century (e.g., 2024)
# %m - Month as a zero-padded decimal number (01 to 12)
# %d - Day of the month (01 to 31)
# %H - Hour in 24-hour format (00 to 23)
# %M - Minute (00 to 59)
# %S - Second (00 to 59)
```

- The `strftime()` method allows you to convert a `datetime` object into a formatted string according to the format you specify.
- Common format codes include `%Y` for the year, `%m` for the month, and `%d` for the day.

Section 6.3: Parsing String to Datetime Object

To parse a string representation of a date back into a `datetime` object, you can use the `strptime()` method. This method allows you to specify the exact format the string is in so that it can be correctly interpreted.

Example:

```
python
Copy code
from datetime import datetime

# Date string in a specific format
date_string = "2024-12-29 15:30:00"

# Parse string to datetime object
parsed_date = datetime.strptime(date_string, "%Y-%m-%d %H:%
M:%S")
print("Parsed Datetime:", parsed_date)
```

```
# Example with other date formats
# "%B" - Full month name
# "%d" - Day of the month
date_string2 = "December 29, 2024"
parsed_date2 = datetime.strptime(date_string2, "%B %d, %Y")
print("Parsed Date (Full Month):", parsed_date2)
```

- `strptime()` converts a string into a `datetime` object based on the format you specify.
- The format codes must match the structure of the date string being parsed. For example, `%Y-%m-%d` works with `2024-12-29`, while `%B %d, %Y` works with `December 29, 2024`.

Chapter 7: Enum

Enums (short for Enumerations) are a way to define a set of symbolic names bound to unique, constant integer values. Python's `enum` module allows for the creation of enumerations, which help make code more readable and maintainable by replacing magic numbers or arbitrary strings with meaningful names.

Section 7.1: Creating an Enum (Python 2.4 through 3.3)

In versions of Python before 3.4, creating an enum required defining a class that inherits from `Enum` in the `enum` module. This allows us to define symbolic names for values.

Example:

```
python
Copy code
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

```
# Accessing enum members
print(Color.RED)         # Output: Color.RED
print(Color.GREEN.name)  # Output: 'GREEN'
print(Color.BLUE.value)  # Output: 3
```

- In this example, `Color` is an enumeration with three members: `RED`, `GREEN`, and `BLUE`, each with unique integer values.

Section 7.2: Iteration

Enums can be iterated over, meaning you can loop through all the members of an enumeration. This is useful when you need to process or display all the symbolic names.

Example:

```
python
Copy code
from enum import Enum

class Day(Enum):
    MONDAY = 1
    TUESDAY = 2
    WEDNESDAY = 3
    THURSDAY = 4
    FRIDAY = 5

# Iterate through enum
for day in Day:
    print(f"{day.name}: {day.value}")
```

- The iteration outputs all members of the `Day` enum, showing both their names and values.

Chapter 8: Set

A `set` is a built-in Python collection type that is unordered, mutable, and contains no duplicate elements. It's an essential data structure for operations like checking membership, removing duplicates from a list, and performing mathematical set operations like union, intersection, and difference.

Section 8.1: Operations on Sets

Python sets support several operations, including adding and removing elements, testing membership, and performing set operations.

Example:

```
python
Copy code
# Create a set
my_set = {1, 2, 3, 4, 5}

# Add an element
my_set.add(6)

# Remove an element
my_set.remove(4)

# Check membership
print(3 in my_set)  # Output: True
print(10 in my_set) # Output: False

# Set operations
set_a = {1, 2, 3}
set_b = {3, 4, 5}

print(set_a.union(set_b))    # Output: {1, 2, 3, 4, 5}
print(set_a.intersection(set_b))  # Output: {3}
print(set_a.difference(set_b))    # Output: {1, 2}
```

- `add()` adds an element to the set.
- `remove()` removes an element from the set.

- `union()`, `intersection()`, and `difference()` are methods for common set operations.

Section 8.2: Get the Unique Elements of a List

One of the most common use cases for sets is extracting unique elements from a list, as sets inherently remove duplicates.

Example:

```
python
Copy code
# List with duplicates
my_list = [1, 2, 2, 3, 4, 4, 5]

# Convert list to set to get unique elements
unique_elements = set(my_list)
print(unique_elements)  # Output: {1, 2, 3, 4, 5}
```

- Converting a list to a set automatically removes any duplicates.

Section 8.3: Set of Sets

A set of sets refers to a set that contains other sets as its elements. However, sets themselves are mutable and therefore cannot contain other sets. To work with sets of sets, you can use **frozensets**, which are immutable sets.

Example:

```
python
Copy code
# Create frozensets
set_a = frozenset([1, 2])
set_b = frozenset([3, 4])

# Create a set of frozensets
set_of_sets = {set_a, set_b}
print(set_of_sets)  # Output: {frozenset({1, 2}), frozenset
```

```
({3, 4})}
```

- `frozenset` is used to create immutable sets, which can be added to a regular set.

Section 8.4: Set Operations Using Methods and Built-ins

Python provides both methods and built-in functions to perform common set operations.

Example:

```
python
Copy code
# Using methods
set_a = {1, 2, 3}
set_b = {3, 4, 5}

# Union
print(set_a | set_b)    # Output: {1, 2, 3, 4, 5}
# Intersection
print(set_a & set_b)    # Output: {3}
# Difference
print(set_a - set_b)    # Output: {1, 2}

# Using methods
print(set_a.union(set_b))      # Output: {1, 2, 3, 4, 5}
print(set_a.intersection(set_b))  # Output: {3}
print(set_a.difference(set_b))    # Output: {1, 2}
```

- `|`, `&`, and `-` are set operators for union, intersection, and difference, respectively.
- `union()`, `intersection()`, and `difference()` are equivalent methods.

Section 8.5: Sets Versus Multisets

A **set** is a collection of unique elements, while a **multiset** allows duplicate elements. Python sets do not support duplicates, but you can simulate multisets using the `collections.Counter` class, which counts the occurrences of elements.

Chapter 9: Simple Mathematical Operators

In Python, mathematical operations can be performed using standard operators and functions. These operators are essential for any arithmetic and scientific computations, ranging from simple addition and subtraction to more complex operations like exponentiation, trigonometric functions, and logarithms.

Section 9.1: Division

Division in Python can be performed using the `/` operator, which returns a floating-point result. Python also supports integer division with the `//` operator, which discards the decimal part and returns an integer.

Example:

```
python
Copy code
# Division (float result)
result = 10 / 3
print(result)  # Output: 3.3333333333333335

# Integer Division (floor division)
result_int = 10 // 3
print(result_int)  # Output: 3
```

- `/` returns a float.
- `//` returns the quotient as an integer, truncating the result.

Section 9.2: Addition

Addition in Python is straightforward and uses the `+` operator. It can be used to add numbers, as well as to concatenate strings or combine lists.

Example:

```
python
Copy code
# Adding two numbers
sum_result = 5 + 3
print(sum_result)  # Output: 8

# Adding strings (concatenation)
greeting = "Hello " + "World!"
print(greeting)  # Output: 'Hello World!'

# Adding lists (concatenation)
list_result = [1, 2] + [3, 4]
print(list_result)  # Output: [1, 2, 3, 4]
```

- The `+` operator can be used for numeric addition, string concatenation, and list merging.

Section 9.3: Exponentiation

Exponentiation is performed using the `**` operator. This operator raises the left operand to the power of the right operand.

Example:

```
python
Copy code
# Exponentiation
result = 2 ** 3
print(result)  # Output: 8
```

- `2 ** 3` means 2 raised to the power of 3, which results in 8.

Section 9.4: Trigonometric Functions

Python's `math` module provides a range of trigonometric functions, including `sin()`, `cos()`, `tan()`, and others. These functions work with radians, so if you need to work with degrees, you'll need to convert them using the `math.radians()` function.

Example:

```
python
Copy code
import math

# Sine function
angle_radians = math.radians(30)  # Convert 30 degrees to r
adians
sin_value = math.sin(angle_radians)
print(sin_value)  # Output: 0.49999999999999994

# Cosine function
cos_value = math.cos(angle_radians)
print(cos_value)  # Output: 0.8660254037844387
```

- Trigonometric functions like `math.sin()` and `math.cos()` expect input in radians.

Section 9.5: Inplace Operations

In Python, inplace operations allow you to modify variables directly without needing to create a new variable. Inplace operations often use the `+=`, `-=` operators, among others.

Example:

```
python
Copy code
# Inplace addition
a = 5
a += 3
print(a)  # Output: 8
```

```
# Inplace multiplication
b = 4
b *= 2
print(b)  # Output: 8
```

- `+=` adds the right operand to the left operand and assigns the result to the left operand, modifying the original variable.

Section 9.6: Subtraction

Subtraction is performed using the `-` operator in Python.

Example:

```
python
Copy code
# Subtraction
result = 10 - 4
print(result)  # Output: 6
```

- The `-` operator simply subtracts the right operand from the left operand.

Section 9.7: Multiplication

Multiplication is performed using the `*` operator in Python. This operator can be used for numeric multiplication, string repetition, and list multiplication.

Example:

```
python
Copy code
# Multiplying numbers
result = 4 * 5
print(result)  # Output: 20

# String repetition
```

```
repeat_string = "Hello" * 3
print(repeat_string)  # Output: 'HelloHelloHello'

# List repetition
repeat_list = [1, 2] * 3
print(repeat_list)  # Output: [1, 2, 1, 2, 1, 2]
```

- The `*` operator is versatile and can multiply numbers, repeat strings, or expand lists.

Section 9.8: Logarithms

Python's `math` module provides functions for logarithmic calculations, such as `log()`, which can compute logarithms with different bases. The default base is `e` (natural logarithm), but you can specify any base.

Example:

```
python
Copy code
import math

# Natural logarithm (base e)
log_value = math.log(10)
print(log_value)  # Output: 2.302585092994046

# Logarithm with a different base (base 10)
log_base10 = math.log(100, 10)
print(log_base10)  # Output: 2.0
```

- `math.log()` computes the natural logarithm by default, and you can specify another base as the second argument.

Section 9.9: Modulus

The modulus operator (`%`) returns the remainder of a division operation. It is often used for determining whether a number is divisible by another or for

cyclic operations.

Example:

```
python
Copy code
# Modulus (remainder)
remainder = 10 % 3
print(remainder)  # Output: 1
```

- In the example, `10 % 3` returns `1`, because 3 fits into 10 three times with a remainder of 1.

Chapter 10: Bitwise Operators

Bitwise operators in Python allow manipulation of individual bits in numbers. These operators are used in a variety of tasks, such as cryptography, network programming, and low-level systems programming.

Section 10.1: Bitwise NOT

The **bitwise NOT** operator (`~`) inverts all the bits of a number, changing `1` to `0` and `0` to `1`.

Example:

```
python
Copy code
# Bitwise NOT
x = 5  # In binary: 101
result = ~x  # Inverts the bits: 010 (which is -6 in decima
l)
print(result)  # Output: -6
```

- `~5` results in `6` because the **bitwise NOT** of a number is equivalent to `(n + 1)`.

Section 10.2: Bitwise XOR (Exclusive OR)

The **bitwise XOR** operator (`^`) compares the bits of two numbers. It returns `1` if the bits are different, and `0` if they are the same.

Example:

```
python
Copy code
# Bitwise XOR
x = 5  # In binary: 101
y = 3  # In binary: 011
result = x ^ y  # XOR: 110 (which is 6 in decimal)
print(result)  # Output: 6
```

- `5 ^ 3` results in `6` because the binary representation of `5` is `101` and `3` is `011`. XORing these gives `110` (which is `6` in decimal).

Section 10.3: Bitwise AND

The **bitwise AND** operator (`&`) compares the bits of two numbers. It returns `1` if both bits are `1`, and `0` otherwise.

Example:

```
python
Copy code
# Bitwise AND
x = 5  # In binary: 101
y = 3  # In binary: 011
result = x & y  # AND: 001 (which is 1 in decimal)
print(result)  # Output: 1
```

- `5 & 3` results in `1` because the binary representation of `5` is `101` and `3` is `011`. ANDing these gives `001` (which is `1` in decimal).

Section 10.4: Bitwise OR

The **bitwise OR** operator (`|`) compares the bits of two numbers. It returns `1` if at least one of the bits is `1`, and `0` only if both bits are `0`.

Example:

```
python
Copy code
# Bitwise OR
x = 5  # In binary: 101
y = 3  # In binary: 011
result = x | y  # OR: 111 (which is 7 in decimal)
print(result)  # Output: 7
```

- `5 | 3` results in `7` because the binary representation of `5` is `101` and `3` is `011`. ORing these gives `111` (which is `7` in decimal).

Section 10.5: Bitwise Left Shift

The **bitwise left shift** operator (`<<`) shifts the bits of a number to the left by a specified number of positions. This operation is equivalent to multiplying the number by `2` for each position shifted.

Example:

```
python
Copy code
# Bitwise Left Shift
x = 5  # In binary: 101
result = x << 1  # Shifts left by 1 bit: 1010 (which is 10
in decimal)
print(result)  # Output: 10
```

- `5 << 1` results in `10` because shifting the bits of `5` left by one position doubles it (`101` becomes `1010`).

Section 10.6: Bitwise Right Shift

The **bitwise right shift** operator (`>>`) shifts the bits of a number to the right by a specified number of positions. This operation is equivalent to dividing the number by `2` for each position shifted.

Example:

```
python
Copy code
# Bitwise Right Shift
x = 5  # In binary: 101
result = x >> 1  # Shifts right by 1 bit: 010 (which is 2 i
n decimal)
print(result)  # Output: 2
```

- `5 >> 1` results in `2` because shifting the bits of `5` right by one position halves it (`101` becomes `010`).

Section 10.7: Inplace Operations

Bitwise operations can be performed inplace, modifying the value of a variable directly using the following operators:

- `&=`, `|=`, `^=`, `<<=`, `>>=`

Example:

```
python
Copy code
# Inplace Bitwise Operations
x = 5  # In binary: 101
x &= 3  # AND operation with 3 (011)
print(x)  # Output: 1

y = 5  # In binary: 101
y |= 3  # OR operation with 3 (011)
print(y)  # Output: 7

z = 5  # In binary: 101
z <<= 1  # Left shift by 1 (101 becomes 1010)
```

```
print(z)  # Output: 10
```

- `x &= 3` modifies `x` to `1` (binary `001`), `y |= 3` modifies `y` to `7` (binary `111`), and `z <<= 1` modifies `z` to `10` (binary `1010`).

Chapter 11: Boolean Operators

Boolean operators in Python are used for logical operations on conditions or boolean values. The primary operators are `and`, `or`, and `not`.

Section 11.1: `and` and `or` are not guaranteed to return a boolean

In Python, the `and` and `or` operators do not always return a boolean value (`True` or `False`). Instead, they return one of the operands, which can be any value (not just boolean).

Example:

```
python
Copy code
# `and` and `or` returning non-boolean values
x = 0
y = 5
result_and = x and y  # Returns 0 (because x is falsy)
result_or = x or y    # Returns 5 (because y is truthy)

print(result_and)  # Output: 0
print(result_or)   # Output: 5
```

- `and` returns the first falsy value, or the last truthy value if both are truthy.
- `or` returns the first truthy value, or the last falsy value if both are falsy.

Section 11.2: A simple example

A simple example shows how the `and`, `or`, and `not` operators work with boolean values.

Example:

```
python
Copy code
a = True
b = False

# `and` operator
print(a and b)  # Output: False

# `or` operator
print(a or b)  # Output: True

# `not` operator
print(not a)   # Output: False
```

- `a and b` returns `False` because both operands are not `True`.
- `a or b` returns `True` because at least one operand is `True`.
- `not a` returns `False` because `a` is `True`.

Section 11.3: Short-circuit evaluation

Python's boolean operators use **short-circuit evaluation**, which means that if the result of the operation can be determined by the first operand, the second operand is not evaluated.

Example:

```
python
Copy code
# Short-circuiting in 'and' and 'or'
x = 0
y = 5

# 'and' short-circuits (x is falsy, so y is not evaluated)
result_and = x and y
print(result_and)  # Output: 0
```

```
# 'or' short-circuits (x is falsy, so y is evaluated)
result_or = x or y
print(result_or)  # Output: 5
```

- The `and` operator short-circuits and returns the first falsy value, without evaluating the second operand.
- The `or` operator short-circuits and returns the first truthy value, but evaluates the second operand if the first is falsy.

Section 11.4: `and`

The `and` operator returns `True` if both operands are `True`; otherwise, it returns `False`.

Example:

```
python
Copy code
print(True and True)   # Output: True
print(True and False)  # Output: False
```

Section 11.5: `or`

The `or` operator returns `True` if at least one of the operands is `True`; otherwise, it returns `False`.

Example:

```
python
Copy code
print(True or False)   # Output: True
print(False or False)  # Output: False
```

Section 11.6: `not`

The `not` operator inverts the boolean value of an operand. It returns `True` if the operand is `False`, and `False` if the operand is `True`.

Example:

```
python
Copy code
print(not True)  # Output: False
print(not False) # Output: True
```

Chapter 12: Operator Precedence

Operator precedence in Python refers to the order in which operators are evaluated in an expression. The precedence determines how the operators are grouped in the absence of parentheses, which affect the order of evaluation.

Section 12.1: Simple Operator Precedence Examples in Python

Python follows specific rules to determine the precedence of operators. Here is a simplified list of the most common operators in order of precedence (from highest to lowest):

1. **Parentheses** `()`
2. **Exponentiation** `*`
3. **Unary plus, Unary minus, and Bitwise NOT** `+x`, `x`, `~x`
4. **Multiplication, Division, Floor Division, and Modulus** `, /, //, %`
5. **Addition and Subtraction** `+`, `
6. **Bitwise Shift Operators** `<<`, `>>`
7. **Bitwise AND** `&`
8. **Bitwise XOR** `^`
9. **Bitwise OR** `|`

10. **Comparison Operators** `==`, `!=`, `<`, `>`, `<=`, `>=`
11. **Boolean NOT** `not`
12. **Boolean AND** `and`
13. **Boolean OR** `or`

When operators of the same precedence appear, Python evaluates them from **left to right** (this is known as **left-associativity**), except for the exponentiation operator (`**`), which is **right-associative**.

Example:

```
python
Copy code
# Operator Precedence Example
result = 3 + 5 * 2
print(result)  # Output: 13
```

In this case, multiplication (`*`) has higher precedence than addition (`+`), so `5 * 2` is evaluated first, and then `3 + 10` is evaluated to give `13`.

Example with Parentheses:

```
python
Copy code
# Using Parentheses to change precedence
result = (3 + 5) * 2
print(result)  # Output: 16
```

Here, the parentheses force addition to occur first, and then the result is multiplied by `2`, yielding `16`.

Chapter 13: Variable Scope and Binding

In Python, variable scope refers to the area in a program where a variable is accessible. The scope of a variable defines how long it exists and how accessible it is during program execution. There are several types of scopes:

- **Local Scope**: Variables defined within a function or block are local to that block.
- **Global Scope**: Variables defined at the top level of a script or module are global.
- **Nonlocal Scope**: A variable that is neither local nor global but is in a higher scope, usually used in nested functions.

Section 13.1: Nonlocal Variables

The `nonlocal` keyword allows you to work with variables in the nearest enclosing scope, excluding the global scope.

Example:

```
python
Copy code
def outer_function():
    x = 10  # x is a local variable to outer_function
    def inner_function():
        nonlocal x  # Refers to the nearest enclosing scope
(outer_function)
        x += 5
    inner_function()
    print(x)  # Output: 15

outer_function()
```

In this example, `x` is a nonlocal variable in `inner_function()`, and modifying it changes the value in the `outer_function()` scope.

Section 13.2: Global Variables

Global variables are defined outside of any function or class and are accessible throughout the program, but they can be modified inside functions using the `global` keyword.

Example:

```
python
Copy code
x = 10  # Global variable

def modify_global():
    global x
    x += 5

modify_global()
print(x)  # Output: 15
```

In this example, `global x` tells Python to refer to the global `x`, and the function modifies it.

Section 13.3: Local Variables

Local variables are defined within a function and are only accessible within that function.

Example:

```
python
Copy code
def my_function():
    x = 5  # Local variable
    print(x)

my_function()
# print(x)  # This would raise an error because x is local
to my_function.
```

- `x` in this example is a local variable, and it is not accessible outside of `my_function()`.

Section 13.4: The `del` Command

The `del` command is used to delete variables or elements from collections like lists and dictionaries.

Example:

```
python
Copy code
x = 10
del x  # Deletes variable x
# print(x)  # This would raise an error because x is delete
d.
```

- You can also use `del` to remove items from a list or dictionary:

```
python
Copy code
lst = [1, 2, 3]
del lst[0]  # Removes the first element of the list
print(lst)  # Output: [2, 3]
```

Section 13.5: Functions Skip Class Scope When Looking Up Names

When Python looks for a variable name, it first checks the local scope (inside the function), then the enclosing scopes, and finally the global scope. If the variable is not found in any of these scopes, an error is raised.

Example:

```
python
Copy code
class MyClass:
    x = 20  # Class variable

    def my_method(self):
        x = 10  # Local variable
```

```
        print(x)  # Prints local variable, not the class va
riable

obj = MyClass()
obj.my_method()  # Output: 10
```

- In this case, the method `my_method` has a local variable `x`, so it prints `10` instead of the class variable `x`.

Section 13.6: Local vs Global Scope

The local scope is specific to the function, and global variables are accessible throughout the script. However, global variables can be shadowed by local variables if they have the same name.

Example:

```
python
Copy code
x = 5  # Global variable

def my_function():
    x = 10  # Local variable
    print(x)

my_function()  # Output: 10
print(x)  # Output: 5 (global variable)
```

- `my_function()` uses a local variable `x`, which shadows the global variable `x` during its execution.

Section 13.7: Binding Occurrence

Binding refers to the association of a variable with a value. A variable is bound to a value when it is assigned. The binding occurrence occurs at the point where a variable is first assigned or modified.

Example:

```
python
Copy code
def my_function():
    x = 5  # Binding x to 5

my_function()
```

- `x` is bound to the value `5` in the local scope of `my_function()`.

Chapter 14: Conditionals

Conditionals in Python allow the program to make decisions based on certain conditions, enabling different execution paths. This chapter covers several fundamental concepts related to conditionals, including the **ternary operator**, the use of `if`, `elif`, `else`, and boolean logic.

Section 14.1: Conditional Expression (or "The Ternary Operator")

The conditional expression (also known as the **ternary operator**) provides a shorthand way of writing an `if-else` statement. It follows the format:

```
python
Copy code
value_if_true if condition else value_if_false
```

Example:

```
python
Copy code
age = 18
status = "Adult" if age >= 18 else "Minor"
```

```
print(status)  # Output: "Adult"
```

In this example, the ternary operator checks if `age` is greater than or equal to 18. If the condition is true, it assigns `"Adult"` to `status`; otherwise, it assigns `"Minor"`.

Section 14.2: `if`, `elif`, and `else`

The `if`, `elif`, and `else` statements allow you to execute blocks of code based on conditions:

- `if`: Evaluates the condition and executes the corresponding block of code if the condition is `True`.
- `elif`: Stands for "else if" and is used to evaluate additional conditions if the `if` condition is `False`.
- `else`: Executes the corresponding block of code if all preceding conditions are `False`.

Example:

```
python
Copy code
age = 20
if age < 18:
    print("Minor")
elif age == 18:
    print("Just an adult")
else:
    print("Adult")
```

- If `age` is less than 18, it prints `"Minor"`.
- If `age` equals 18, it prints `"Just an adult"`.
- If `age` is greater than 18, it prints `"Adult"`.

Section 14.3: Truth Values

In Python, conditional expressions evaluate **truth values**. These truth values are determined based on whether a condition is `True` or `False`. Python uses the following rules:

- **False values**: `None`, `False`, `0`, empty sequences (e.g., `[]`, `()`, `""`), and empty dictionaries `{}`.
- **True values**: Everything else, including non-zero numbers, non-empty strings, lists, tuples, etc.

Example:

```
python
Copy code
if []:  # Empty list is considered False
    print("This won't print")
else:
    print("This will print")
```

Section 14.4: Boolean Logic Expressions

Boolean logic expressions allow you to combine multiple conditions using logical operators:

- `and`: Returns `True` if both operands are `True`.
- `or`: Returns `True` if at least one operand is `True`.
- `not`: Reverses the truth value (i.e., `True` becomes `False`, and vice versa).

Example:

```
python
Copy code
x = 10
y = 5
if x > 5 and y < 10:
    print("Both conditions are true")
```

- The condition `x > 5 and y < 10` evaluates to `True` because both sub-conditions are true.

Section 14.5: Using the `cmp` Function to Get the Comparison Result of Two Objects

In earlier versions of Python (before 3.x), the `cmp()` function was used to compare two objects and return:

- `0` if the objects are equal
- `1` if the first object is greater
- `1` if the first object is smaller

However, `cmp()` is no longer available in Python 3.x. Instead, you can use comparison operators like `==`, `>`, and `<`.

Example (in Python 2.x):

```
python
Copy code
result = cmp(3, 4)  # Returns -1
print(result)
```

In Python 3.x, you can simply use:

```
python
Copy code
result = 3 < 4  # Returns True
```

Section 14.6: Else Statement

The `else` statement provides an alternative block of code that is executed when the `if` or `elif` conditions are not true.

Example:

```
python
Copy code
number = 10
if number % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

In this example, if the condition `number % 2 == 0` is `True`, `"Even number"` will be printed; otherwise, `"Odd number"` will be printed.

Section 14.7: Testing if an Object is `None` and Assigning It

In Python, `None` is a special value representing the absence of a value. To test if an object is `None`, you use the `is` operator:

Example:

```
python
Copy code
x = None
if x is None:
    print("x is None")
else:
    print("x is not None")
```

- The condition `x is None` checks whether `x` is the special `None` object.

Section 14.8: If Statement

The `if` statement is the most basic form of conditional execution. It evaluates a condition and executes a block of code if the condition is true.

Example:

```
python
Copy code
```

```
temperature = 30
if temperature > 25:
    print("It's hot outside!")
```

- If `temperature` is greater than 25, the message `"It's hot outside!"` will be printed.

Chapter 15: Comparisons

In this chapter, we explore how Python handles comparisons between values and objects. Comparisons are essential for controlling the flow of execution in programs and for evaluating conditions in various situations. This chapter covers chain comparisons, the difference between `is` and `==`, comparison operators, and comparing objects.

Section 15.1: Chain Comparisons

Python supports **chain comparisons**, where you can evaluate multiple comparisons in a single statement. This is equivalent to combining conditions using logical operators but in a more readable way.

Syntax:

```
python
Copy code
x < y < z
```

This can be interpreted as:

```
python
Copy code
x < y and y < z
```

Example:

```
python
Copy code
x = 3
y = 5
z = 10

if x < y < z:
    print("x is less than y and y is less than z")
```

In this case, both `x < y` and `y < z` are true, so the output is:

```
csharp
Copy code
x is less than y and y is less than z
```

Section 15.2: Comparison by `is` vs `==`

In Python, `==` and `is` are both used for comparison, but they have different meanings:

- `==`: Checks if the values of two objects are equal.
- `is`: Checks if two objects refer to the same location in memory (i.e., if they are the same object).

Example:

```
python
Copy code
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print(a == b)  # True, because they have the same value
print(a is b)  # False, because they are different objects
```

```
in memory
print(a is c)  # True, because c refers to the same object
as a
```

- `a == b` returns `True` because both lists have the same content.
- `a is b` returns `False` because `a` and `b` are two different objects in memory.
- `a is c` returns `True` because `c` refers to the same list object as `a`.

Section 15.3: Greater Than or Less Than

Python provides comparison operators to check if one value is greater than or less than another:

- `>`: Greater than
- `<`: Less than
- `>=`: Greater than or equal to
- `<=`: Less than or equal to

Example:

```
python
Copy code
x = 10
y = 5

print(x > y)  # True, because 10 is greater than 5
print(x < y)  # False, because 10 is not less than 5
print(x >= 10)  # True, because 10 is greater than or equal
to 10
print(y <= 10)  # True, because 5 is less than or equal to
10
```

Section 15.4: Not Equal To

The `!=` operator checks if two values are **not equal**. It returns `True` if the values are different and `False` if they are the same.

Example:

```
python
Copy code
a = 10
b = 20

print(a != b)  # True, because 10 is not equal to 20
```

Section 15.5: Equal To

The `==` operator checks if two values are **equal**. It returns `True` if the values are the same and `False` if they are different.

Example:

```
python
Copy code
a = "hello"
b = "hello"
c = "world"

print(a == b)  # True, because both strings are the same
print(a == c)  # False, because the strings are different
```

Section 15.6: Comparing Objects

When comparing objects in Python, we can either use the `==` operator to compare their values or the `is` operator to check if they refer to the same memory location. Python provides a variety of comparison methods depending on the object type, including the `__eq__`, `__lt__`, `__le__`, `__gt__`, `__ge__`, and `__ne__` methods that can be overridden for custom objects.

Example: Comparing Custom Objects

```
python
Copy code
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        return self.name == other.name and self.age == othe
r.age

# Create two objects with the same data
person1 = Person("Alice", 30)
person2 = Person("Alice", 30)

print(person1 == person2)  # True, because the data is the
same

# Create two objects with different data
person3 = Person("Bob", 25)

print(person1 == person3)  # False, because the data is dif
ferent
```

In this example, the `__eq__` method is defined to compare two `Person` objects based on their `name` and `age` attributes.

Chapter 16: Loops

Loops are a core part of programming in Python, allowing you to execute a block of code multiple times under specific conditions. This chapter explores the various types of loops in Python and their unique behaviors.

Section 16.1: Break and Continue in Loops

The `break` and `continue` statements are used within loops to alter their normal flow:

- `break`: Terminates the loop entirely, stopping any further iterations.
- `continue`: Skips the current iteration and moves to the next one.

Example:

```
python
Copy code
# Break example
for i in range(5):
    if i == 3:
        break
    print(i)

# Output:
# 0
# 1
# 2

# Continue example
for i in range(5):
    if i == 3:
        continue
    print(i)

# Output:
# 0
# 1
# 2
# 4
```

In the first example, the loop breaks when `i` equals 3. In the second example, the loop skips over `i == 3` and continues with the next iteration.

Section 16.2: For Loops

The `for` loop in Python is used to iterate over a sequence (like a list, tuple, dictionary, or string).

Syntax:

```
python
Copy code
for item in iterable:
    # Do something with item
```

Example:

```
python
Copy code
for char in "hiringhustle":
    print(char)
```

This loop prints each character in the string "hiringhustle":

```
css
Copy code
h
i
r
i
n
g
h
u
s
t
l
e
```

Section 16.3: Iterating Over Lists

You can iterate over lists using a `for` loop. The loop will go through each element in the list one by one.

Example:

```
python
Copy code
names = ["Alice", "Bob", "Charlie"]

for name in names:
    print(name)
```

This loop prints:

```
Copy code
Alice
Bob
Charlie
```

Section 16.4: Loops with an "Else" Clause

The `else` clause can be used in loops to specify a block of code to execute after the loop finishes normally (i.e., it did not terminate via a `break` statement).

Example:

```
python
Copy code
for char in "hiringhustle":
    if char == "u":
        break
else:
    print("Completed the loop without breaking.")
```

Since the loop breaks when `char` equals `"u"`, the `else` block is skipped. If no `break` occurs, the `else` block will run.

Section 16.5: The Pass Statement

The `pass` statement is a placeholder. It does nothing and is used when a statement is syntactically required but you don't want to implement anything in that spot.

Example:

```
python
Copy code
for i in range(5):
    if i == 3:
        pass  # Placeholder for future code
    print(i)
```

This loop runs as usual, but when `i` equals 3, the `pass` statement is executed, which does nothing.

Section 16.6: Iterating Over Dictionaries

When iterating over a dictionary, you can access its keys, values, or both using methods like `.keys()`, `.values()`, and `.items()`.

Example:

```
python
Copy code
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Iterating over keys
for key in my_dict:
    print(key)

# Iterating over values
for value in my_dict.values():
    print(value)

# Iterating over key-value pairs
for key, value in my_dict.items():
```

```
    print(key, value)
```

Output:

```
sql
Copy code
name
age
city

Alice
25
New York

name Alice
age 25
city New York
```

Section 16.7: The "Half Loop" do-while

Python does not have a built-in `do-while` loop like some other languages, but you can mimic this behavior using a `while` loop that runs at least once and checks the condition at the end.

Example:

```
python
Copy code
i = 0
while True:
    print(i)
    i += 1
    if i >= 3:
        break
```

This loop prints:

```
Copy code
0
1
2
```

Section 16.8: Looping and Unpacking

In Python, you can unpack elements in a loop, especially useful when working with tuples or lists.

Example:

```
python
Copy code
pairs = [(1, 'a'), (2, 'b'), (3, 'c')]

for num, letter in pairs:
    print(num, letter)
```

This loop prints:

```
css
Copy code
1 a
2 b
3 c
```

Section 16.9: Iterating Different Portions of a List with Different Step Size

You can specify a step size in a loop to iterate over parts of a list, which is done using the `range()` function.

Example:

```
python
Copy code
names = ["Alice", "Bob", "Charlie", "David", "Eve"]

# Iterate over every second name
for name in names[::2]:
    print(name)
```

Output:

```
Copy code
Alice
Charlie
Eve
```

Section 16.10: While Loop

A `while` loop continues to execute a block of code as long as a specified condition is true.

Syntax:

```
python
Copy code
while condition:
    # Do something
```

Example:

```
python
Copy code
i = 0
while i < 3:
```

```
    print("hiringhustle"[i])  # Prints each character until
i reaches 3
    i += 1
```

This loop prints:

```
css
Copy code
h
i
r
```

Chapter 17: Arrays

In Python, arrays are often used for storing and manipulating collections of data in a compact way. While Python lists are frequently used, the `array` module allows for more efficient handling of homogeneous data (same data type) in an array-like structure.

Section 17.1: Access Individual Elements Through Indexes

You can access elements in an array using indices, just like lists. The index starts at `0` and increases by `1` for each subsequent element.

Example:

```
python
Copy code
import array

arr = array.array('u', 'hiringhustle')  # 'u' is the type c
ode for unicode characters
print(arr[0])  # Accessing first element
```

```
print(arr[1])  # Accessing second element
```

Output:

```
css
Copy code
h
i
```

Section 17.2: Basic Introduction to Arrays

Arrays are similar to lists but require elements to be of the same type. The `array` module in Python provides efficient storage for basic types.

Example:

```
python
Copy code
import array

# Creating an array of integers
arr = array.array('i', [1, 2, 3, 4, 5])
print(arr)
```

Section 17.3: Append Any Value to the Array Using `append()` Method

You can add a new item to the end of the array using the `append()` method.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3])
arr.append(4)  # Adds 4 to the end
```

```
print(arr)
```

Output:

```
c
Copy code
array('i', [1, 2, 3, 4])
```

Section 17.4: Insert Value in an Array Using `insert()` Method

The `insert()` method allows you to add an element at a specific position in the array.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3])
arr.insert(1, 10)  # Inserts 10 at index 1
print(arr)
```

Output:

```
c
Copy code
array('i', [1, 10, 2, 3])
```

Section 17.5: Extend Python Array Using `extend()` Method

You can add multiple elements to the array using the `extend()` method.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3])
arr.extend([4, 5, 6])  # Adds the elements of the list to t
he array
print(arr)
```

Output:

```
c
Copy code
array('i', [1, 2, 3, 4, 5, 6])
```

Section 17.6: Add Items from List into Array Using `fromlist()` Method

This method adds elements from a Python list to the end of the array.

Example:

```
python
Copy code
arr = array.array('i', [1, 2])
arr.fromlist([3, 4, 5])  # Adds the list elements to the ar
ray
print(arr)
```

Output:

```
c
Copy code
array('i', [1, 2, 3, 4, 5])
```

Section 17.7: Remove Any Array Element Using `remove()` Method

The `remove()` method removes the first occurrence of a specified element.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4, 5])
arr.remove(3)  # Removes the element 3
print(arr)
```

Output:

```
c
Copy code
array('i', [1, 2, 4, 5])
```

Section 17.8: Remove Last Array Element Using `pop()` Method

The `pop()` method removes and returns the last element of the array.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4])
last_item = arr.pop()  # Removes the last item (4)
print(last_item)  # Prints: 4
print(arr)
```

Output:

```
c
Copy code
```

```
4
array('i', [1, 2, 3])
```

Section 17.9: Fetch Any Element Through Its Index Using `index()` Method

The `index()` method returns the index of the first occurrence of a specified value.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4])
index_of_three = arr.index(3)  # Finds index of 3
print(index_of_three)
```

Output:

```
Copy code
2
```

Section 17.10: Reverse a Python Array Using `reverse()` Method

The `reverse()` method reverses the order of the elements in the array.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4])
arr.reverse()
```

```
print(arr)
```

Output:

```
c
Copy code
array('i', [4, 3, 2, 1])
```

Section 17.11: Get Array Buffer Information Through `buffer_info()` Method

This method returns a tuple with the address of the array's buffer and the number of elements.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4])
buffer_info = arr.buffer_info()
print(buffer_info)
```

Output:

```
scss
Copy code
(42949672960, 4)
```

Section 17.12: Check for Number of Occurrences of an Element Using `count()` Method

The `count()` method returns the number of occurrences of a specified element.

Example:

```
python
Copy code
arr = array.array('i', [1, 2, 3, 1, 1, 4])
count_of_ones = arr.count(1)
print(count_of_ones)
```

Output:

```
Copy code
3
```

Section 17.13: Convert Array to String Using `tostring()` Method

The `tostring()` method converts the array to a string representation. **Note:** `tostring()` is deprecated, and it's better to use `tobytes()`.

Example:

```
python
Copy code
arr = array.array('u', 'hiringhustle')
string_rep = arr.tobytes()  # Using the correct method
print(string_rep)
```

Section 17.14: Convert Array to a Python List with Same Elements Using `tolist()` Method

The `tolist()` method converts an array to a regular Python list.

Example:

```
python
Copy code
```

```
arr = array.array('i', [1, 2, 3, 4])
list_rep = arr.tolist()
print(list_rep)
```

Output:

```
csharp
Copy code
[1, 2, 3, 4]
```

Section 17.15: Append a String to Char Array Using `fromstring()` Method

The `fromstring()` method allows you to append characters from a string to a character array.

Example:

```
python
Copy code
arr = array.array('u', 'hiringhustle')
arr.fromstring('ManoharJoshi')  # Adds the string to the ar
ray
print(arr)
```

Output:

```
c
Copy code
array('u', 'hiringhustleManoharJoshi')
```

```
python
Copy code
```

```
from collections import Counter

# Multiset using Counter
multiset = Counter([1, 1, 2, 2, 2, 3])
print(multiset)  # Output: Counter({2: 3, 1: 2, 3: 1})

# Accessing the count of an element
print(multiset[2])  # Output: 3
```

- `Counter` helps create a multiset-like structure, where elements can appear multiple times.

Chapter 18: Multidimensional Arrays

Multidimensional arrays are arrays that contain other arrays as elements. These are often used to represent matrices, grids, or more complex structures such as images.

Section 18.1: Lists in Lists

In Python, you can create a list of lists, which can act like a 2D array. This allows you to represent rows and columns of data.

Example:

```
python
Copy code
# List of lists (2D array)
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Accessing elements
print(matrix[0][0])  # First row, first element
```

```
print(matrix[1][2])  # Second row, third element
```

Output:

```
Copy code
1
6
```

Section 18.2: Lists in Lists in Lists

You can have lists inside lists inside other lists, forming a multidimensional array of more than two dimensions.

Example:

```
python
Copy code
# 3D Array (List of lists of lists)
cube = [
    [
        [1, 2],
        [3, 4]
    ],
    [
        [5, 6],
        [7, 8]
    ]
]

# Accessing elements
print(cube[0][1][1])  # First 2D array, second row, second
element
```

Output:

```
Copy code
4
```

Chapter 19: Dictionary

Dictionaries are a collection of key-value pairs. They are unordered, mutable, and indexed by keys, allowing for fast lookups, inserts, and updates.

Section 19.1: Introduction to Dictionary

A dictionary is a collection of key-value pairs, where each key is unique. You can access values by referring to their keys.

Example:

```
python
Copy code
# Creating a dictionary
student = {
    'name': 'John',
    'age': 25,
    'major': 'Computer Science'
}

# Accessing values
print(student['name'])  # Output: John
```

Section 19.2: Avoiding KeyError Exceptions

When accessing dictionary values, you may encounter a `KeyError` if the key doesn't exist. To avoid this, you can use methods like `get()`.

Example:

```
python
Copy code
```

```
# Using get() method to avoid KeyError
student = {
    'name': 'John',
    'age': 25
}

print(student.get('major', 'Not Found'))  # Output: Not Fou
nd
```

Section 19.3: Iterating Over a Dictionary

You can iterate over the keys, values, or key-value pairs of a dictionary using loops.

Example:

```
python
Copy code
# Iterating over dictionary
student = {
    'name': 'John',
    'age': 25,
    'major': 'Computer Science'
}

# Iterating through keys and values
for key, value in student.items():
    print(f'{key}: {value}')
```

Output:

```
makefile
Copy code
name: John
age: 25
```

```
major: Computer Science
```

Section 19.4: Dictionary with Default Values

You can use the `defaultdict` from the `collections` module to provide a default value for keys that don't exist.

Example:

```
python
Copy code
from collections import defaultdict

# Creating a defaultdict with int as default value
inventory = defaultdict(int)

inventory['apples'] += 5  # Adds 5 apples
inventory['bananas'] += 3  # Adds 3 bananas

print(inventory)  # Output: defaultdict(<class 'int'>, {'ap
ples': 5, 'bananas': 3})
```

Section 19.5: Merging Dictionaries

You can merge dictionaries using the `update()` method or the `|` operator in Python 3.9+.

Example:

```
python
Copy code
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}

# Using update() method
dict1.update(dict2)
```

```
print(dict1)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Using | operator in Python 3.9+
merged_dict = dict1 | dict2
print(merged_dict)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd':
4}
```

Section 19.6: Accessing Keys and Values

You can access the keys and values of a dictionary using the `keys()` and `values()` methods.

Example:

```
python
Copy code
student = {
    'name': 'John',
    'age': 25,
    'major': 'Computer Science'
}

# Accessing keys
print(student.keys())  # Output: dict_keys(['name', 'age',
'major'])

# Accessing values
print(student.values())  # Output: dict_values(['John', 25,
'Computer Science'])
```

Section 19.7: Accessing Values of a Dictionary

You can access the values of a dictionary using the `get()` method or by directly referencing the key.

Example:

```
python
Copy code
student = {
    'name': 'John',
    'age': 25,
    'major': 'Computer Science'
}

# Using get() method
print(student.get('age'))  # Output: 25

# Directly referencing the key
print(student['major'])  # Output: Computer Science
```

Section 19.8: Creating a Dictionary

Dictionaries can be created using curly braces `{}` or the `dict()` constructor.

Example:

```
python
Copy code
# Creating a dictionary using curly braces
student = {'name': 'John', 'age': 25}

# Creating a dictionary using dict()
student2 = dict(name='John', age=25)

print(student)  # Output: {'name': 'John', 'age': 25}
print(student2)  # Output: {'name': 'John', 'age': 25}
```

Section 19.9: Creating an Ordered Dictionary

Python 3.7+ maintains the insertion order of keys. However, for earlier versions, you can use `OrderedDict` from the `collections` module to maintain order.

Example:

```
python
Copy code
from collections import OrderedDict

# Creating an ordered dictionary
ordered_dict = OrderedDict([
    ('first', 1),
    ('second', 2),
    ('third', 3)
])

print(ordered_dict)  # Output: OrderedDict([('first', 1),
('second', 2), ('third', 3)])
```

Section 19.10: Unpacking Dictionaries Using the `*` Operator

You can use the `**` operator to unpack dictionaries when passing them as arguments to functions or when combining dictionaries.

Example:

```
python
Copy code
dict1 = {'name': 'John'}
dict2 = {'age': 25}

# Unpacking dictionaries
merged = {**dict1, **dict2}
print(merged)  # Output: {'name': 'John', 'age': 25}
```

Section 19.11: The Trailing Comma

The trailing comma can be used in dictionaries, especially when adding new key-value pairs.

Example:

```
python
Copy code
student = {
    'name': 'John',
    'age': 25,
    'major': 'Computer Science',
}

# Trailing comma allows you to add new keys easily
print(student)
```

Section 19.12: The `dict()` Constructor

The `dict()` constructor is another way to create dictionaries, which can be useful for more complex dictionary creation.

Example:

```
python
Copy code
# Creating dictionary using dict constructor
student = dict(name='John', age=25, major='Computer Scienc
e')
print(student)
```

Section 19.13: Dictionaries Example

Here's a simple example of using dictionaries:

Example:

```
python
Copy code
student = {
```

```
    'name': 'John',
    'age': 25,
    'major': 'Computer Science'
}

# Update a value
student['age'] = 26

# Add a new key-value pair
student['year'] = 'Senior'

print(student)
```

Section 19.14: All Combinations of Dictionary Values

You can find all combinations of dictionary values using `itertools.product()` if needed.

Example:

```
python
Copy code
from itertools import product

dict1 = {'a': [1, 2], 'b': [3, 4]}

# Getting all combinations of dictionary values
combinations = list(product(*dict1.values()))
print(combinations)  # Output: [(1, 3), (1, 4), (2, 3), (2,
4)]
```

Chapter 20: List

Lists in Python are versatile and commonly used data structures that store ordered collections of items. Lists can hold elements of different data types and

allow for easy manipulation and access to these elements.

Section 20.1: List Methods and Supported Operators

Python provides a wide range of built-in methods and operators for working with lists. Some common methods include:

- **append()**: Adds an element to the end of the list.
- **insert()**: Inserts an element at a specific index.
- **remove()**: Removes the first occurrence of a value.
- **pop()**: Removes and returns an element at a specific index.
- **extend()**: Adds multiple elements from another iterable.
- **clear()**: Removes all elements from the list.
- **index()**: Returns the index of the first occurrence of a value.
- **count()**: Returns the number of occurrences of a value.
- **sort()**: Sorts the list in ascending order.
- **reverse()**: Reverses the list.

Example:

```
python
Copy code
fruits = ['apple', 'banana', 'cherry']

# Using append() to add an element
fruits.append('orange')

# Using insert() to add an element at a specific position
fruits.insert(1, 'kiwi')

# Using remove() to remove an element
fruits.remove('banana')

print(fruits)  # Output: ['apple', 'kiwi', 'cherry', 'orang
```

```
e']
```

Section 20.2: Accessing List Values

You can access list values by their index. List indexing starts from 0 for the first element. Negative indices can be used to access elements from the end of the list.

Example:

```
python
Copy code
fruits = ['apple', 'banana', 'cherry']

print(fruits[0])   # Output: apple
print(fruits[-1])  # Output: cherry
```

Section 20.3: Checking if List is Empty

You can check whether a list is empty using the `len()` function or directly by evaluating the list in a boolean context.

Example:

```
python
Copy code
fruits = []

if not fruits:
    print("The list is empty.")  # Output: The list is empt
y.
```

Section 20.4: Iterating Over a List

You can iterate over the elements of a list using loops like `for` or list comprehensions.

Example:

```
python
Copy code
fruits = ['apple', 'banana', 'cherry']

# Using a for loop
for fruit in fruits:
    print(fruit)
```

Output:

```
Copy code
apple
banana
cherry
```

Section 20.5: Checking Whether an Item is in a List

You can check if an item exists in a list using the `in` operator.

Example:

```
python
Copy code
fruits = ['apple', 'banana', 'cherry']

if 'banana' in fruits:
    print("Banana is in the list.")  # Output: Banana is in
the list.
```

Section 20.6: Any and All

The `any()` function returns `True` if at least one element in the list is true. The `all()` function returns `True` only if all elements in the list are true.

Example:

```
python
Copy code
numbers = [0, 1, 2, 3]

print(any(numbers))  # Output: True (because 1, 2, 3 are tr
uthy)
print(all(numbers))  # Output: False (because 0 is falsy)
```

Section 20.7: Reversing List Elements

You can reverse a list using the `reverse()` method or slicing.

Example:

```
python
Copy code
fruits = ['apple', 'banana', 'cherry']

# Using reverse()
fruits.reverse()
print(fruits)  # Output: ['cherry', 'banana', 'apple']

# Using slicing
print(fruits[::-1])  # Output: ['apple', 'banana', 'cherr
y']
```

Section 20.8: Concatenate and Merge Lists

You can concatenate (combine) lists using the `+` operator or the `extend()` method.

Example:

```
python
Copy code
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Using + operator
merged = list1 + list2
print(merged)  # Output: [1, 2, 3, 4, 5, 6]

# Using extend()
list1.extend(list2)
print(list1)  # Output: [1, 2, 3, 4, 5, 6]
```

Section 20.9: Length of a List

You can get the length (number of elements) of a list using the `len()` function.

Example:

```
python
Copy code
fruits = ['apple', 'banana', 'cherry']

print(len(fruits))  # Output: 3
```

Section 20.10: Remove Duplicate Values in List

You can remove duplicates from a list by converting it to a set and back to a list, or by using a loop.

Example:

```
python
Copy code
```

```
fruits = ['apple', 'banana', 'cherry', 'banana', 'apple']

# Using set to remove duplicates
fruits = list(set(fruits))
print(fruits)  # Output: ['apple', 'banana', 'cherry']

# Using loop (preserving order)
fruits = ['apple', 'banana', 'cherry', 'banana', 'apple']
unique_fruits = []
for fruit in fruits:
    if fruit not in unique_fruits:
        unique_fruits.append(fruit)
print(unique_fruits)  # Output: ['apple', 'banana', 'cherr
y']
```

Section 20.11: Comparison of Lists

Lists can be compared using relational operators, and they will be evaluated element by element.

Example:

```
python
Copy code
list1 = [1, 2, 3]
list2 = [1, 2, 3]

print(list1 == list2)  # Output: True

list3 = [4, 5, 6]
print(list1 < list3)  # Output: True
```

Section 20.12: Accessing Values in Nested List

You can access values in a nested list (a list inside another list) by using multiple indices.

Example:

```
python
Copy code
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Accessing nested elements
print(nested_list[0][1])  # Output: 2
print(nested_list[2][0])  # Output: 7
```

Section 20.13: Initializing a List to a Fixed Number of Elements

You can initialize a list with a fixed number of elements by repeating a value using multiplication.

Example:

```
python
Copy code
# Initializing a list with 5 zeros
zeros = [0] * 5
print(zeros)  # Output: [0, 0, 0, 0, 0]
```

Chapter 21: List Comprehensions

List comprehensions provide a concise way to create and manipulate lists. They allow you to perform complex operations in a single line of code, which improves readability and efficiency. This chapter explores various types of comprehensions, including conditional comprehensions, dictionary comprehensions, and other advanced use cases.

Section 21.1: List Comprehensions

A list comprehension offers a compact way to process all or part of the elements in a sequence and return a list with the results. The syntax is:

```
python
Copy code
[expression for item in iterable]
```

Example:

```
python
Copy code
# Simple list comprehension to square numbers
numbers = [1, 2, 3, 4]
squares = [n**2 for n in numbers]
print(squares)  # Output: [1, 4, 9, 16]
```

Section 21.2: Conditional List Comprehensions

You can add an `if` condition to a list comprehension to filter elements based on a condition.

Example:

```
python
Copy code
# List comprehension with a condition to get even numbers
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = [n for n in numbers if n % 2 == 0]
print(even_numbers)  # Output: [2, 4, 6]
```

You can also use `else` in a list comprehension to apply different operations depending on the condition.

Example:

```
python
Copy code
```

```
# Conditional operation with 'else'
numbers = [1, 2, 3, 4, 5]
results = [n**2 if n % 2 == 0 else n**3 for n in numbers]
print(results)  # Output: [1, 4, 27, 16, 125]
```

Section 21.3: Avoid Repetitive and Expensive Operations Using Conditional Clause

By placing conditions within list comprehensions, you can prevent repetitive or expensive operations on elements that do not meet the condition.

Example:

```
python
Copy code
# Avoid repetitive expensive operation by skipping unnecess
ary values
numbers = [1, 2, 3, 4, 5]
squared_odd_numbers = [n**2 for n in numbers if n % 2 != 0]
print(squared_odd_numbers)  # Output: [1, 9, 25]
```

This helps optimize your code by only performing operations on necessary elements.

Section 21.4: Dictionary Comprehensions

In addition to list comprehensions, Python supports dictionary comprehensions, which create a dictionary from an iterable.

Syntax:

```
python
Copy code
{key_expression: value_expression for item in iterable}
```

Example:

```
python
Copy code
# Dictionary comprehension to create a square map
numbers = [1, 2, 3, 4]
square_dict = {n: n**2 for n in numbers}
print(square_dict)  # Output: {1: 1, 2: 4, 3: 9, 4: 16}
```

Section 21.5: List Comprehensions with Nested Loops

You can use multiple `for` clauses in list comprehensions to handle nested iterations. This is particularly useful for working with multidimensional data structures.

Example:

```
python
Copy code
# List comprehension with nested loops to flatten a list of
lists
nested_lists = [[1, 2], [3, 4], [5, 6]]
flattened = [item for sublist in nested_lists for item in s
ublist]
print(flattened)  # Output: [1, 2, 3, 4, 5, 6]
```

Section 21.6: Generator Expressions

Generator expressions are similar to list comprehensions but return a generator instead of a list, which is more memory efficient, especially for large datasets.

Syntax:

```
python
Copy code
```

```
(expression for item in iterable)
```

Example:

```
python
Copy code
# Generator expression to calculate squares
numbers = [1, 2, 3, 4]
squares_gen = (n**2 for n in numbers)

# Using the generator
for square in squares_gen:
    print(square)
```

Section 21.7: Set Comprehensions

Set comprehensions are similar to list comprehensions but create a set. This ensures that duplicate elements are automatically removed.

Syntax:

```
python
Copy code
{expression for item in iterable}
```

Example:

```
python
Copy code
# Set comprehension to get unique squares
numbers = [1, 2, 3, 4, 4]
squares_set = {n**2 for n in numbers}
print(squares_set)  # Output: {16, 1, 4, 9}
```

Section 21.8: Refactoring Filter and Map to List Comprehensions

You can refactor common functions like `filter()` and `map()` into list comprehensions to improve readability.

Example:

```
python
Copy code
# Using filter
numbers = [1, 2, 3, 4, 5]
even_numbers_filter = list(filter(lambda x: x % 2 == 0, num
bers))

# Refactoring to list comprehension
even_numbers_comprehension = [x for x in numbers if x % 2 =
= 0]
print(even_numbers_comprehension)  # Output: [2, 4]
```

Section 21.9: Comprehensions Involving Tuples

You can also use comprehensions to create tuples. This is useful when you need to generate ordered pairs or perform transformations.

Example:

```
python
Copy code
# Tuple comprehension
numbers = [1, 2, 3, 4]
tuples = tuple((n, n**2) for n in numbers)
print(tuples)  # Output: ((1, 1), (2, 4), (3, 9), (4, 16))
```

Section 21.10: Counting Occurrences Using Comprehension

List comprehensions can be used to count occurrences of specific elements or to filter based on their frequency.

Example:

```
python
Copy code
# Counting occurrences of even numbers in a list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
count_even = sum(1 for n in numbers if n % 2 == 0)
print(count_even)  # Output: 5
```

Section 21.11: Changing Types in a List

Comprehensions can also be used to convert elements in a list to different types.

Example:

```
python
Copy code
# Converting strings to integers
strings = ["1", "2", "3", "4"]
integers = [int(s) for s in strings]
print(integers)  # Output: [1, 2, 3, 4]
```

Section 21.12: Nested List Comprehensions

List comprehensions can be nested within each other to handle more complex data transformations.

Example:

```
python
Copy code
# Nested list comprehension to extract characters from mult
```

```
iple words
words = ["apple", "banana", "cherry"]
letters = [letter for word in words for letter in word]
print(letters)  # Output: ['a', 'p', 'p', 'l', 'e', 'b',
'a', 'n', 'a', 'n', 'a', 'c', 'h', 'e', 'r', 'r', 'y']
```

Section 21.13: Iterate Two or More Lists Simultaneously within List Comprehension

You can iterate over multiple lists at the same time in a list comprehension by using the `zip()` function.

Example:

```
python
Copy code
# Iterating over two lists simultaneously
names = ["Alice", "Bob", "Charlie"]
scores = [85, 90, 88]

result = [(name, score) for name, score in zip(names, score
s)]
print(result)  # Output: [('Alice', 85), ('Bob', 90), ('Cha
rlie', 88)]
```

Chapter 22: List Slicing (Selecting Parts of Lists)

List slicing in Python is a powerful technique that allows you to extract a portion of a list by specifying a start, stop, and optional step value. It is performed using the slicing syntax:

```
python
Copy code
list[start:stop:step]
```

Section 22.1: Using the Third "Step" Argument

The step argument allows you to skip elements at a specified interval while slicing a list.

Syntax:

```
python
Copy code
list[start:stop:step]
```

Example:

```
python
Copy code
# Using the step argument to select every second element
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
step_slice = numbers[::2]
print(step_slice)  # Output: [0, 2, 4, 6, 8]
```

Using a negative step allows you to reverse the slicing direction.

Example:

```
python
Copy code
# Using a negative step to select elements in reverse order
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
reverse_step_slice = numbers[::-2]
print(reverse_step_slice)  # Output: [9, 7, 5, 3, 1]
```

Section 22.2: Selecting a Sublist from a List

You can use slicing to extract a sublist by specifying start and stop indices.

Example:

```
python
Copy code
# Selecting a sublist from index 2 to 5
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sublist = numbers[2:6]  # Includes index 2, excludes index
6
print(sublist)  # Output: [2, 3, 4, 5]
```

If you omit the `start` or `stop` index, slicing defaults to the beginning or the end of the list.

Example:

```
python
Copy code
# Slicing from the beginning to index 4
start_slice = numbers[:5]
print(start_slice)  # Output: [0, 1, 2, 3, 4]

# Slicing from index 5 to the end
end_slice = numbers[5:]
print(end_slice)  # Output: [5, 6, 7, 8, 9]
```

Section 22.3: Reversing a List with Slicing

A common use of slicing is to reverse a list using a step value of `-1`.

Example:

```
python
Copy code
# Reversing a list using slicing
numbers = [0, 1, 2, 3, 4, 5]
reversed_list = numbers[::-1]
```

```
print(reversed_list)  # Output: [5, 4, 3, 2, 1, 0]
```

Reversing a list can be useful for iterating or manipulating data in reverse order.

Section 22.4: Shifting a List Using Slicing

Slicing can also be used to shift elements in a list by reordering them.

Example:

```
python
Copy code
# Shifting a list by moving the first three elements to the
end
numbers = [0, 1, 2, 3, 4, 5, 6]
shifted_list = numbers[3:] + numbers[:3]
print(shifted_list)  # Output: [3, 4, 5, 6, 0, 1, 2]
```

This technique is particularly useful for creating rotated versions of lists.

Chapter 23: `groupby()`

The `groupby()` function from the `itertools` module allows developers to group elements of an iterable based on a specified key or property. This function creates an iterator that produces consecutive keys and groups from the input iterable.

Parameter Details

- `iterable`: Any Python iterable (e.g., list, tuple).
- `key`: A function or criteria based on which the grouping is performed.

Section 23.1: Example 4

Example 1: Using Tuples in the Iterable

Grouping items by their first element:

```
python
Copy code
from itertools import groupby

things = [
    ("animal", "bear"),
    ("animal", "duck"),
    ("plant", "cactus"),
    ("vehicle", "harley"),
    ("vehicle", "speed boat"),
    ("vehicle", "school bus")
]

dic = {}
f = lambda x: x[0]  # Group by the first element of each tu
ple
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)

print(dic)
```

Results:

```
python
Copy code
{
    'animal': [('animal', 'bear'), ('animal', 'duck')],
    'plant': [('plant', 'cactus')],
    'vehicle': [('vehicle', 'harley'), ('vehicle', 'speed b
oat'), ('vehicle', 'school bus')]
}
```

Example 2: Using Lists in the Iterable

The behavior remains the same when the input contains lists instead of tuples.

```
python
Copy code
things = [
    ["animal", "bear"],
    ["animal", "duck"],
    ["vehicle", "harley"],
    ["plant", "cactus"],
    ["vehicle", "speed boat"],
    ["vehicle", "school bus"]
]

dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)

print(dic)
```

Results:

```
python
Copy code
{
    'animal': [['animal', 'bear'], ['animal', 'duck']],
    'plant': [['plant', 'cactus']],
    'vehicle': [['vehicle', 'harley'], ['vehicle', 'speed b
oat'], ['vehicle', 'school bus']]
}
```

Section 23.2: Example 2

When no `key` is provided, the default behavior is grouping consecutive identical elements as keys.

Example:

```
python
Copy code
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('pe
rsons', 'man', 'woman')])
dic = {}

for k, v in c:
    dic[k] = list(v)

print(dic)
```

Results:

```
python
Copy code
{
    1: [1, 1],
    2: [2],
    3: [3],
    ('persons', 'man', 'woman'): [('persons', 'man', 'woma
n')],
    'cow': ['cow'],
    'dog': ['dog'],
    10: [10],
    11: [11],
    'goat': ['goat']
}
```

Section 23.3: Example 3

When the input data is not sorted, only the last occurrence of a key is considered.

Example: Without Sorting

```
python
Copy code
from itertools import groupby

list_things = [
    'goat', 'dog', 'donkey', 'mulato', 'cow', 'cat',
    ('persons', 'man', 'woman'), 'wombat', 'mongoose',
    'malloo', 'camel'
]

c = groupby(list_things, key=lambda x: x[0])
dic = {}

for k, v in c:
    dic[k] = list(v)

print(dic)
```

Results:

```
python
Copy code
{
    'c': ['camel'],
    'd': ['dog', 'donkey'],
    'g': ['goat'],
    'm': ['mongoose', 'malloo'],
    'persons': [('persons', 'man', 'woman')],
    'w': ['wombat']
}
```

Example: With Sorting

To include all elements under their respective keys, the input must be sorted first.

```
python
Copy code
sorted_list = sorted(list_things, key=lambda x: x[0])
print(sorted_list)  # ['cow', 'cat', 'camel', 'dog', 'donke
y', 'goat', ...]

c = groupby(sorted_list, key=lambda x: x[0])
dic = {}

for k, v in c:
    dic[k] = list(v)

print(dic)
```

Results:

```
python
Copy code
{
    'c': ['cow', 'cat', 'camel'],
    'd': ['dog', 'donkey'],
    'g': ['goat'],
    'm': ['mulato', 'mongoose', 'malloo'],
    'persons': [('persons', 'man', 'woman')]
}
```

Key Takeaways

1. **Sorting Before Grouping**: Always sort the iterable by the same key function used in `groupby()` for consistent results.
2. **Default Behavior**: Without a specified key, consecutive identical elements are grouped together.
3. **Versatile Usage**: `groupby()` works seamlessly with tuples, lists, and other iterables.

Chapter 24: Linked Lists

Linked lists are a fundamental data structure used to store data in a sequential manner. Unlike arrays, linked lists consist of nodes where each node contains:

1. Data.
2. A reference to the next node in the sequence.

Section 24.1: Single Linked List Example

Implementation of a Singly Linked List

Here's an example of creating and manipulating a singly linked list in Python:

```
python
Copy code
class Node:
    """A class to represent a single node in a linked lis
t."""
    def __init__(self, data=None):
        self.data = data
        self.next = None

class LinkedList:
    """A class to represent a singly linked list."""
    def __init__(self):
        self.head = None

    def append(self, data):
        """Add a new node at the end of the linked list."""
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
```

```
        current.next = new_node

    def display(self):
        """Print all the elements in the linked list."""
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        print(" -> ".join(map(str, elements)))

    def delete(self, key):
        """Delete the first occurrence of the key in the li
nked list."""
        current = self.head
        if current and current.data == key:
            self.head = current.next
            current = None
            return
        prev = None
        while current and current.data != key:
            prev = current
            current = current.next
        if current is None:
            return
        prev.next = current.next
        current = None

# Example Usage
linked_list = LinkedList()
linked_list.append(10)
linked_list.append(20)
linked_list.append(30)
linked_list.display()  # Output: 10 -> 20 -> 30

linked_list.delete(20)
linked_list.display()  # Output: 10 -> 30
```

Chapter 25: Linked List Node

Nodes are the building blocks of a linked list. Each node contains the data and a reference to the next node.

Section 25.1: Write a Simple Linked List Node in Python

Here's how to define a basic linked list node:

```
python
Copy code
class Node:
    def __init__(self, data=None):
        """Initialize a node with data and a next pointe
r."""
        self.data = data
        self.next = None

# Example Usage
node1 = Node(5)
node2 = Node(10)
node1.next = node2  # Linking node1 to node2

print(node1.data)  # Output: 5
print(node1.next.data)  # Output: 10
```

Chapter 26: Filter

The `filter()` function is a built-in function in Python used to filter elements from an iterable based on a specified condition.

Section 26.1: Basic Use of `filter()`

Example

Using `filter()` with a lambda function:

```
python
Copy code
# Filter even numbers from a list
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6]
```

Section 26.2: Filter Without Function

Example

Filtering out `None` or falsy values from a list:

```
python
Copy code
items = [0, 1, None, 2, '', 3, False]
filtered_items = list(filter(None, items))
print(filtered_items)  # Output: [1, 2, 3]
```

Section 26.3: Filter as Short-Circuit Check

Using `filter()` to quickly evaluate conditions:

```
python
Copy code
# Check if any number in the list is divisible by 5
numbers = [1, 2, 3, 4, 5, 6, 10]
divisible_by_5 = list(filter(lambda x: x % 5 == 0, number
s))
print(divisible_by_5)  # Output: [5, 10]
```

Section 26.4: Complementary Function: `filterfalse` and `ifilterfalse`

`filterfalse` is a complementary function to `filter()` from the `itertools` module. It returns elements for which the condition is `False`.

Example

```
python
Copy code
from itertools import filterfalse

# Remove even numbers
numbers = [1, 2, 3, 4, 5, 6]
odd_numbers = list(filterfalse(lambda x: x % 2 == 0, number
s))
print(odd_numbers)  # Output: [1, 3, 5]
```

Chapter 27: Heapq

The `heapq` module in Python provides an implementation of the heap queue algorithm, also known as the priority queue algorithm. This module is useful for finding the smallest or largest items in a collection efficiently.

Section 27.1: Largest and Smallest Items in a Collection

Example

Find the three largest and smallest items in a list:

```
python
Copy code
import heapq

# List of items
items = [1, 8, 3, 2, 7, 4, 10, 6]

# Find the 3 largest and smallest items
largest = heapq.nlargest(3, items)
smallest = heapq.nsmallest(3, items)
```

```
print("Largest items:", largest)  # Output: [10, 8, 7]
print("Smallest items:", smallest)  # Output: [1, 2, 3]
```

Section 27.2: Smallest Item in a Collection

Using `heapq.heappop()` to retrieve the smallest item while maintaining the heap property:

```
python
Copy code
import heapq

# Create a list and heapify it
items = [6, 3, 8, 1, 4, 7]
heapq.heapify(items)

# Pop the smallest item
smallest = heapq.heappop(items)
print("Smallest item:", smallest)  # Output: 1
print("Remaining heap:", items)  # Output: [3, 4, 8, 6, 7]
```

Chapter 28: Tuple

Tuples are immutable sequences in Python, often used to store collections of related data.

Section 28.1: Tuple

Example

Creating a tuple with names:

```
python
Copy code
names = ("hiringhustle", "ManoharJoshi", "RishiKumar", "Mah
eshBabu")
print(names)  # Output: ('hiringhustle', 'ManoharJoshi', 'R
```

```
ishiKumar', 'MaheshBabu')
```

Section 28.2: Tuples Are Immutable

Tuples cannot be modified after creation:

```
python
Copy code
names = ("hiringhustle", "ManoharJoshi", "RishiKumar", "Mah
eshBabu")
try:
    names[0] = "NewName"  # Attempt to modify
except TypeError as e:
    print(e)  # Output: 'tuple' object does not support ite
m assignment
```

Section 28.3: Packing and Unpacking Tuples

Packing

Packing values into a tuple:

```
python
Copy code
packed = ("hiringhustle", "ManoharJoshi", "RishiKumar", "Ma
heshBabu")
print(packed)  # Output: ('hiringhustle', 'ManoharJoshi',
'RishiKumar', 'MaheshBabu')
```

Unpacking

Unpacking values into variables:

```
python
Copy code
```

```
a, b, c, d = ("hiringhustle", "ManoharJoshi", "RishiKumar",
"MaheshBabu")
print(a, b, c, d)  # Output: hiringhustle ManoharJoshi Rish
iKumar MaheshBabu
```

Section 28.4: Built-in Tuple Functions

Example

Using `len()` and `count()`:

```
python
Copy code
names = ("hiringhustle", "ManoharJoshi", "RishiKumar", "Mah
eshBabu")

# Length of tuple
print(len(names))  # Output: 4

# Count occurrences of an element
print(names.count("hiringhustle"))  # Output: 1
```

Section 28.5: Tuples Are Element-wise Hashable and Equatable

Example

Comparing tuples:

```
python
Copy code
tuple1 = ("hiringhustle", "ManoharJoshi")
tuple2 = ("hiringhustle", "ManoharJoshi")
tuple3 = ("RishiKumar", "MaheshBabu")

print(tuple1 == tuple2)  # Output: True
```

```
print(tuple1 == tuple3)  # Output: False
```

Section 28.6: Indexing Tuples

Example

Accessing elements by index:

```
python
Copy code
names = ("hiringhustle", "ManoharJoshi", "RishiKumar", "Mah
eshBabu")

# Access first and last elements
print(names[0])  # Output: hiringhustle
print(names[-1])  # Output: MaheshBabu
```

Section 28.7: Reversing Elements

Example

Reversing a tuple:

```
python
Copy code
names = ("hiringhustle", "ManoharJoshi", "RishiKumar", "Mah
eshBabu")

# Reverse the tuple
reversed_names = names[::-1]
print(reversed_names)  # Output: ('MaheshBabu', 'RishiKuma
r', 'ManoharJoshi', 'hiringhustle')
```

Chapter 29: Basic Input and Output

Section 29.1: Using the `print` Function

The `print` function outputs data to the standard output.

Example

```
python
Copy code
print("Hello, World!")  # Output: Hello, World!
```

Section 29.2: Input from a File

Using `open()` to read a file:

```
python
Copy code
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # Reads each line and removes
trailing newline
```

Section 29.3: Read from `stdin`

Reading from standard input:

```
python
Copy code
import sys

for line in sys.stdin:
    print(line.strip())  # Process each line entered
```

Section 29.4: Using `input()` and `raw_input()`

In Python 3, `input()` reads user input as a string:

```
python
Copy code
name = input("Enter your name: ")
print(f"Hello, {name}!")  # Output: Hello, <name>!
```

Section 29.5: Function to Prompt User for a Number

Example

```
python
Copy code
def prompt_for_number():
    while True:
        try:
            number = int(input("Enter a number: "))
            return number
        except ValueError:
            print("That's not a valid number!")

num = prompt_for_number()
print(f"You entered: {num}")
```

Section 29.6: Printing a String Without a Newline at the End

Use `end=''` in the `print` function:

```
python
Copy code
print("Hello", end='')
print(" World!")  # Output: Hello World!
```

Chapter 30: Files & Folders I/O

Section 30.1: File Modes

Common modes:

- `'r'`: Read
- `'w'`: Write (overwrite if file exists)
- `'a'`: Append
- `'b'`: Binary mode

Example

```
python
Copy code
# Writing to a file
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

Section 30.2: Reading a File Line-by-Line

```
python
Copy code
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Section 30.3: Iterate Files (Recursively)

Using `os` to iterate through directories:

```
python
Copy code
import os
```

```
for root, dirs, files in os.walk("."):
    for file in files:
        print(os.path.join(root, file))
```

Section 30.4: Getting the Full Contents of a File

```
python
Copy code
with open("example.txt", "r") as file:
    content = file.read()
print(content)
```

Section 30.5: Writing to a File

```
python
Copy code
with open("example.txt", "w") as file:
    file.write("New content!")
```

Section 30.6: Check Whether a File or Path Exists

Using `os.path.exists`:

```
python
Copy code
import os

if os.path.exists("example.txt"):
    print("File exists!")
else:
    print("File does not exist.")
```

Section 30.7: Random File Access Using `mmap`

```
python
Copy code
import mmap

with open("example.txt", "r+b") as file:
    mmapped_file = mmap.mmap(file.fileno(), 0)
    print(mmapped_file.readline().strip())
    mmapped_file.close()
```

Section 30.8: Replacing Text in a File

```
python
Copy code
with open("example.txt", "r") as file:
    content = file.read()

content = content.replace("old_text", "new_text")

with open("example.txt", "w") as file:
    file.write(content)
```

Section 30.9: Checking if a File is Empty

```
python
Copy code
import os

if os.stat("example.txt").st_size == 0:
    print("File is empty!")
else:
    print("File is not empty.")
```

Section 30.10: Read a File Between a Range of Lines

```
python
Copy code
with open("example.txt", "r") as file:
    lines = file.readlines()
    for line in lines[10:20]:  # Reading lines 10-20
        print(line.strip())
```

Section 30.11: Copy a Directory Tree

Using `shutil.copytree`:

```
python
Copy code
import shutil

shutil.copytree("source_folder", "destination_folder")
```

Section 30.12: Copying Contents of One File to Another

```
python
Copy code
with open("source.txt", "r") as src, open("destination.tx
t", "w") as dest:
    dest.write(src.read())
```

Chapter 31: os.path

Section 31.1: Join Paths

Use `os.path.join()` to construct file paths in an OS-independent way.

Example

```
python
Copy code
import os

path = os.path.join("folder", "subfolder", "file.txt")
print(path)  # Output: folder/subfolder/file.txt (Windows u
ses backslashes)
```

Section 31.2: Path Component Manipulation

Using `os.path.basename` and `os.path.dirname`:

```
python
Copy code
import os

path = "/folder/subfolder/file.txt"
print(os.path.basename(path))  # Output: file.txt
print(os.path.dirname(path))   # Output: /folder/subfolder
```

Section 31.3: Get the Parent Directory

Retrieve the parent directory using `os.path.abspath` and `os.path.dirname`:

```
python
Copy code
import os

path = "/folder/subfolder/file.txt"
parent_dir = os.path.dirname(path)
```

```
print(parent_dir)  # Output: /folder/subfolder
```

Section 31.4: Check if the Given Path Exists

Using `os.path.exists`:

```
python
Copy code
import os

path = "example.txt"
if os.path.exists(path):
    print("Path exists.")
else:
    print("Path does not exist.")
```

Section 31.5: Check Path Type

Check if a path is a file, directory, or symbolic link:

```
python
Copy code
import os

path = "example.txt"
if os.path.isfile(path):
    print("It's a file.")
elif os.path.isdir(path):
    print("It's a directory.")
elif os.path.islink(path):
    print("It's a symbolic link.")
```

Section 31.6: Absolute Path from Relative Path

Convert a relative path to an absolute path:

```
python
Copy code
import os

relative_path = "./folder/file.txt"
absolute_path = os.path.abspath(relative_path)
print(absolute_path)  # Output: Full absolute path
```

Chapter 32: Iterables and Iterators

Section 32.1: Iterator vs Iterable vs Generator

- **Iterable**: An object capable of returning its members one at a time (e.g., lists, strings).
- **Iterator**: An object with a `__next__()` method to fetch the next item.
- **Generator**: A type of iterator created with a function and the `yield` keyword.

Example

```
python
Copy code
# Iterable
my_list = [1, 2, 3]

# Iterator
my_iter = iter(my_list)
print(next(my_iter))  # Output: 1

# Generator
def my_gen():
    yield 1
    yield 2
    yield 3

gen = my_gen()
```

```
print(next(gen))  # Output: 1
```

Section 32.2: Extract Values One by One

```
python
Copy code
my_list = [10, 20, 30]
my_iter = iter(my_list)

for value in my_iter:
    print(value)  # Output: 10, 20, 30
```

Section 32.3: Iterating Over Entire Iterable

Using `for` to iterate over an iterable:

```
python
Copy code
my_list = [5, 10, 15]
for val in my_list:
    print(val)  # Output: 5, 10, 15
```

Section 32.4: Verify Only One Element in Iterable

Using `any()` and `all()` functions:

```
python
Copy code
numbers = [0, 1, 2]

# Check if any number is non-zero
print(any(numbers))  # Output: True

# Check if all numbers are non-zero
```

```
print(all(numbers))  # Output: False
```

Section 32.5: What Can Be Iterable

Examples of iterable objects:

```
python
Copy code
# Strings
for char in "abc":
    print(char)  # Output: a, b, c

# Tuples
for val in (1, 2, 3):
    print(val)  # Output: 1, 2, 3
```

Section 32.6: Iterator Isn't Reentrant!

An iterator can only be traversed once:

```
python
Copy code
numbers = iter([1, 2, 3])

print(list(numbers))  # Output: [1, 2, 3]
print(list(numbers))  # Output: []
```

Chapter 33: Functions

Section 33.1: Defining and Calling Simple Functions

Functions are reusable blocks of code that perform a specific task.

Example

```
python
Copy code
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))  # Output: Hello, Alice!
```

Section 33.2: Defining a Function with an Arbitrary Number of Arguments

Use `*args` to accept a variable number of positional arguments.

Example

```
python
Copy code
def sum_numbers(*args):
    return sum(args)

print(sum_numbers(1, 2, 3))  # Output: 6
print(sum_numbers(10, 20))   # Output: 30
```

Section 33.3: Lambda (Inline/Anonymous) Functions

Lambda functions are small, anonymous functions defined using the `lambda` keyword.

Example

```
python
Copy code
add = lambda x, y: x + y
print(add(5, 3))  # Output: 8

squared = list(map(lambda x: x**2, [1, 2, 3]))
```

```
print(squared)  # Output: [1, 4, 9]
```

Section 33.4: Defining a Function with Optional Arguments

Provide default values to arguments.

Example

```
python
Copy code
def greet(name, message="Welcome"):
    return f"{message}, {name}!"

print(greet("Alice"))             # Output: Welcome, Alice!
print(greet("Bob", "Hi there"))   # Output: Hi there, Bob!
```

Section 33.5: Defining a Function with Optional Mutable Arguments

Avoid mutable defaults to prevent unexpected behavior.

Example

```
python
Copy code
def append_to_list(value, lst=None):
    if lst is None:
        lst = []
    lst.append(value)
    return lst

print(append_to_list(1))  # Output: [1]
print(append_to_list(2))  # Output: [2]
```

Section 33.6: Argument Passing and Mutability

Mutable arguments (e.g., lists, dictionaries) can be modified inside functions.

Example

```
python
Copy code
def modify_list(lst):
    lst.append(4)

my_list = [1, 2, 3]
modify_list(my_list)
print(my_list)  # Output: [1, 2, 3, 4]
```

Section 33.7: Returning Values from Functions

Functions can return a value using the `return` statement.

Example

```
python
Copy code
def square(num):
    return num * num

print(square(4))  # Output: 16
```

Section 33.8: Closure

Closures capture variables from their containing function's scope.

Example

```
python
Copy code
def multiplier(factor):
```

```
    def multiply(number):
        return number * factor
    return multiply

double = multiplier(2)
print(double(5))  # Output: 10
```

Section 33.9: Forcing the Use of Named Parameters

Place `*` before parameters to enforce named arguments.

Example

```
python
Copy code
def greet(*, name, message):
    return f"{message}, {name}!"

print(greet(name="Alice", message="Hello"))  # Output: Hell
o, Alice!
```

Section 33.10: Nested Functions

Functions can be defined inside other functions.

Example

```
python
Copy code
def outer_function():
    def inner_function():
        return "Inner!"
    return inner_function()

print(outer_function())  # Output: Inner!
```

Chapter 35: Functional Programming in Python

Section 35.1: Lambda Function

Lambda functions are used for concise, single-expression functions.

Example

```
python
Copy code
squared = lambda x: x * x
print(squared(5))  # Output: 25
```

Section 35.2: Map Function

The `map` function applies a function to all items in an input iterable.

Example

```
python
Copy code
nums = [1, 2, 3]
squared = list(map(lambda x: x**2, nums))
print(squared)  # Output: [1, 4, 9]
```

Section 35.3: Reduce Function

The `reduce` function applies a rolling computation.

Example

```
python
Copy code
from functools import reduce

nums = [1, 2, 3, 4]
```

```
result = reduce(lambda x, y: x * y, nums)
print(result)  # Output: 24
```

Section 35.4: Filter Function

The `filter` function filters elements based on a condition.

Example

```
python
Copy code
nums = [1, 2, 3, 4, 5]
even = list(filter(lambda x: x % 2 == 0, nums))
print(even)  # Output: [2, 4]
```

Chapter 37: Decorators

Section 37.1: Decorator Function

A decorator is a function that takes another function as input and returns a modified or enhanced version of that function.

Example

```
python
Copy code
def decorator(func):
    def wrapper():
        print("Before the function call.")
        func()
        print("After the function call.")
    return wrapper

@decorator
def say_hello():
```

```
    print("Hello!")

say_hello()
# Output:
# Before the function call.
# Hello!
# After the function call.
```

Section 37.2: Decorator Class

Decorators can also be implemented as classes using the `__call__` method.

Example

```
python
Copy code
class Decorator:
    def __init__(self, func):
        self.func = func

    def __call__(self):
        print("Class-based decorator: Before the function c
all.")
        self.func()
        print("Class-based decorator: After the function ca
ll.")

@Decorator
def greet():
    print("Hello from a class-based decorator!")

greet()
# Output:
# Class-based decorator: Before the function call.
# Hello from a class-based decorator!
# Class-based decorator: After the function call.
```

Section 37.3: Decorator with Arguments (Decorator Factory)

A decorator factory allows passing arguments to a decorator.

Example

```
python
Copy code
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def say_hi():
    print("Hi!")

say_hi()
# Output:
# Hi!
# Hi!
# Hi!
```

Section 37.4: Making a Decorator Look Like the Decorated Function

Use `functools.wraps` to preserve the original function's metadata.

Example

```
python
Copy code
from functools import wraps
```

```
def decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Before the function call.")
        result = func(*args, **kwargs)
        print("After the function call.")
        return result
    return wrapper

@decorator
def add(a, b):
    return a + b

print(add(2, 3))  # Output: 5
print(add.__name__)  # Output: add
```

Section 37.5: Using a Decorator to Time a Function

Measure the execution time of a function with a decorator.

Example

```
python
Copy code
import time

def timing_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} executed in {end - start:.4
f} seconds.")
        return result
    return wrapper
```

```
@timing_decorator
def slow_function():
    time.sleep(2)
    print("Done!")

slow_function()
# Output:
# Done!
# slow_function executed in 2.0001 seconds.
```

Section 37.6: Create Singleton Class with a Decorator

Enforce a single instance of a class using a decorator.

Example

```
python
Copy code
def singleton(cls):
    instances = {}

    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class SingletonClass:
    def __init__(self):
        print("Instance created.")

obj1 = SingletonClass()  # Output: Instance created.
obj2 = SingletonClass()
print(obj1 is obj2)  # Output: True
```

Chapter 38: Classes

Section 38.1: Introduction to Classes

A class is a blueprint for creating objects. Classes encapsulate data for the object and methods to manipulate that data.

Example

```
python
Copy code
class MyClass:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, {self.name}!")

# Create an instance
obj = MyClass("HiringHustle")
obj.greet()
# Output: Hello, HiringHustle!
```

Section 38.2: Bound, Unbound, and Static Methods

- **Bound Methods**: Automatically pass the instance (`self`) as the first argument.
- **Unbound Methods**: Require manual passing of the instance.
- **Static Methods**: Do not require the instance or class as the first argument.

Example

```
python
Copy code
```

```
class Example:
    def bound_method(self):
        print("Bound method called.")

    @staticmethod
    def static_method():
        print("Static method called.")

# Bound method
obj = Example()
obj.bound_method()

# Static method
Example.static_method()
# Output:
# Bound method called.
# Static method called.
```

Section 38.3: Basic Inheritance

Inheritance allows a class (child) to inherit attributes and methods from another class (parent).

Example

```
python
Copy code
class Parent:
    def show(self):
        print("Parent class method.")

class Child(Parent):
    def display(self):
        print("Child class method.")

child = Child()
child.show()  # Inherited from Parent
```

```
child.display()
# Output:
# Parent class method.
# Child class method.
```

Section 38.4: Monkey Patching

Monkey patching is dynamically modifying a class or module at runtime.

Example

```
python
Copy code
class Sample:
    def method(self):
        print("Original method.")

# Monkey patch the method
def patched_method():
    print("Patched method.")

Sample.method = patched_method

obj = Sample()
obj.method()
# Output: Patched method.
```

Section 38.5: New-style vs. Old-style Classes

New-style classes inherit from `object` (Python 3+), while old-style classes do not (Python 2).

Example (New-Style)

```
python
Copy code
```

```
class NewStyleClass(object):
    pass
```

Example (Old-Style)

```
python
Copy code
class OldStyleClass:
    pass
```

Section 38.6: Class Methods: Alternate Initializers

Class methods are defined using the `@classmethod` decorator and can be used as alternate constructors.

Example

```
python
Copy code
class HiringHustle:
    def __init__(self, name):
        self.name = name

    @classmethod
    def from_name(cls, name):
        return cls(name)

obj = HiringHustle.from_name("MaheshBabu")
print(obj.name)
# Output: MaheshBabu
```

Section 38.7: Multiple Inheritance

Python supports multiple inheritance, where a class can inherit from multiple parent classes.

Example

```
python
Copy code
class Parent1:
    def greet(self):
        print("Hello from Parent1.")

class Parent2:
    def greet(self):
        print("Hello from Parent2.")

class Child(Parent1, Parent2):
    pass

obj = Child()
obj.greet()  # Follows method resolution order (MRO)
# Output: Hello from Parent1.
```

Section 38.8: Properties

Properties allow class attributes to have getters, setters, and deleters, providing controlled access to private variables.

Using `@property` Decorator

The `@property` decorator turns a method into a "getter." Additional decorators, `@<property_name>.setter` and `@<property_name>.deleter`, are used for setting and deleting.

Example: Property with Getter, Setter, and Deleter

```
python
Copy code
class HiringHustle:
    def __init__(self, name):
        self._name = name  # Private variable
```

```
    @property
    def name(self):
        print("Getting name...")
        return self._name

    @name.setter
    def name(self, value):
        print("Setting name...")
        self._name = value

    @name.deleter
    def name(self):
        print("Deleting name...")
        del self._name

# Create instance
obj = HiringHustle("RishiKumar")

# Access property
print(obj.name)  # Getter

# Modify property
obj.name = "MaheshBabu"  # Setter
print(obj.name)

# Delete property
del obj.name
# Output:
# Getting name...
# RishiKumar
# Setting name...
# Getting name...
# MaheshBabu
# Deleting name...
```

Why Use Properties?

1. **Encapsulation**: Control access to instance variables.
2. **Validation**: Add logic when setting or getting values.
3. **Readability**: Access like an attribute instead of a method.

Section 38.10: Class and Instance Variables

Class variables are shared across all instances of a class, while **instance variables** are unique to each instance.

Class Variables

Defined directly in the class body and shared by all instances.

Instance Variables

Defined inside methods (like `__init__`) and are specific to each instance.

Example

```
python
Copy code
class HiringHustle:
    company_name = "HiringHustle"  # Class variable

    def __init__(self, employee_name):
        self.employee_name = employee_name  # Instance vari
able

# Access class variable
print(HiringHustle.company_name)  # HiringHustle

# Create instance
emp1 = HiringHustle("MaheshBabu")
emp2 = HiringHustle("RishiKumar")

# Access instance variables
print(emp1.employee_name)  # MaheshBabu
print(emp2.employee_name)  # RishiKumar
```

```
# Modify class variable
HiringHustle.company_name = "HH Tech"
print(emp1.company_name)  # HH Tech
print(emp2.company_name)  # HH Tech
```

Section 38.11: Class Composition

Class Composition is a design principle where a class is composed of objects from other classes. Instead of using inheritance (which models "is-a" relationships), composition models "has-a" relationships. This approach allows objects to be more flexible and modular, as a class can combine behaviors from multiple components without directly inheriting from them.

Why Use Composition?

- **Reusability**: You can reuse the components (objects of other classes) in multiple classes.
- **Flexibility**: Composition allows you to change or extend the behavior of objects at runtime.
- **Decoupling**: Each component class is independent and can be changed or updated without affecting other parts of the program.

Example of Class Composition

In this example, the `Car` class has components `Engine` and `Wheel` rather than inheriting from them.

```
python
Copy code
class Engine:
    def __init__(self, engine_type):
        self.engine_type = engine_type

    def start(self):
        print(f"{self.engine_type} engine started.")

class Wheel:
```

```
    def __init__(self, wheel_type):
        self.wheel_type = wheel_type

    def rotate(self):
        print(f"{self.wheel_type} wheel is rotating.")

class Car:
    def __init__(self, engine_type, wheel_type):
        self.engine = Engine(engine_type)  # Composed of an
Engine
        self.wheel = Wheel(wheel_type)     # Composed of a
Wheel

    def drive(self):
        self.engine.start()
        self.wheel.rotate()

# Create a Car object with its own engine and wheels
my_car = Car("V8", "Alloy")
my_car.drive()
```

Output:

```
csharp
Copy code
V8 engine started.
Alloy wheel is rotating.
```

Key Points of Composition

1. **Behavior Combination**: A class can combine different behaviors (from different objects).
2. **Encapsulation**: The composed classes are encapsulated inside the main class and do not expose internal details.
3. **Loose Coupling**: Changes in a composed class (e.g., `Wheel`) don't directly affect the main class (`Car`), making it easier to maintain and extend.

Comparing Composition vs. Inheritance

- **Inheritance**: Represents a relationship where one class is a specialized version of another class.
- **Composition**: Represents a relationship where one class is made up of one or more other classes (objects).

Section 38.12: Listing All Class Members

In Python, you can list all members (attributes, methods, and other objects) of a class using built-in functions like `dir()`, `vars()`, or `__dict__`. These provide insights into what attributes and methods a class or object has, which can be useful for introspection, debugging, or dynamically working with objects.

Using `dir()`

The `dir()` function returns a list of all attributes and methods of an object or class, including built-in ones (those that are part of the class or its inheritance chain).

Example:

```
python
Copy code
class Dog:
    species = 'Canis familiaris'

    def __init__(self, name):
        self.name = name

    def bark(self):
        return f"{self.name} says woof!"

# List all members of the Dog class
print(dir(Dog))
```

Output:

```
css
Copy code
['__doc__', '__module__', 'bark', 'name', 'species', '__dict__', '__weakref__']
```

Here, the `dir()` function lists all attributes and methods in the `Dog` class, including the special methods (like `__doc__` and `__module__`), instance variables (`name`), and class variables (`species`).

Using `vars()`

The `vars()` function returns the `__dict__` attribute of an object, which is a dictionary that contains the instance variables (but does not list the methods or inherited attributes).

Example:

```
python
Copy code
class Dog:
    def __init__(self, name):
        self.name = name
        self.age = 3

dog = Dog("Buddy")
print(vars(dog))
```

Output:

```
arduino
Copy code
{'name': 'Buddy', 'age': 3}
```

In this example, `vars(dog)` returns the instance variables of `dog`, which are `name` and `age`.

Using `__dict__`

The `__dict__` attribute of a class or instance is a dictionary containing all of its attributes. It can be used directly to access both instance variables and class variables.

Example:

```
python
Copy code
class Dog:
    species = 'Canis familiaris'

    def __init__(self, name):
        self.name = name

dog = Dog("Buddy")
print(dog.__dict__)
```

Output:

```
arduino
Copy code
{'name': 'Buddy'}
```

In this case, `dog.__dict__` only shows the instance variables (`name`), excluding class variables like `species`.

Summary of Methods to List Class Members

- `dir()`: Lists all attributes and methods of an object or class, including built-in ones.
- `vars()`: Returns the `__dict__` of an object, which contains instance variables.
- `__dict__`: Direct access to the dictionary of instance or class variables.

Section 38.13: Singleton Class

A **Singleton** is a design pattern that ensures a class has only one instance and provides a global point of access to that instance. This pattern is often used when only one object is needed to coordinate actions, such as managing a connection pool, configuration settings, or logging.

How to Implement a Singleton

A common way to implement a Singleton in Python is by overriding the `__new__` method, which controls the object creation process. You can check if an instance already exists and return it if so.

Example:

```
python
Copy code
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(c
ls)
        return cls._instance

# Create two instances
s1 = Singleton()
s2 = Singleton()

# Check if both are the same instance
print(s1 is s2)  # True
```

Output:

```
graphql
Copy code
True
```

In this example, `s1` and `s2` are the same instance, demonstrating that only one instance of `Singleton` is created.

Explanation:

- The `__new__` method is responsible for creating a new instance of the class.
- The `_instance` class variable is used to store the single instance of the class.
- If an instance already exists, it returns that instance instead of creating a new one.

Section 38.14: Descriptors and Dotted Lookups

A **descriptor** is an object that defines how attributes are accessed or modified in another object. It is a powerful feature in Python that allows custom behavior for attribute access, assignment, and deletion.

Descriptors are implemented by creating a class that defines one or more of the following methods:

- `__get__(self, instance, owner)`: Called to retrieve an attribute's value.
- `__set__(self, instance, value)`: Called to set an attribute's value.
- `__delete__(self, instance)`: Called to delete an attribute.

Descriptors are used in the implementation of Python's built-in features like properties, methods, and class variables.

Example of a Simple Descriptor

```
python
Copy code
class MyDescriptor:
    def __get__(self, instance, owner):
        return "Accessing the attribute"

    def __set__(self, instance, value):
        print(f"Setting the attribute to {value}")

class MyClass:
    my_attr = MyDescriptor()
```

```
# Create an instance of MyClass
obj = MyClass()

# Access the attribute
print(obj.my_attr)  # Calls MyDescriptor.__get__

# Set the attribute
obj.my_attr = 42  # Calls MyDescriptor.__set__
```

Output:

```
css
Copy code
Accessing the attribute
Setting the attribute to 42
```

In this example:

- The `MyDescriptor` class implements the descriptor protocol with the `__get__` and `__set__` methods.
- When the `my_attr` attribute is accessed or modified, the corresponding method in `MyDescriptor` is invoked.

Dotted Lookups

Dotted lookups allow access to attributes in an object using the dot notation (e.g., `obj.attribute`). When Python performs a dotted lookup, it first checks if the attribute exists in the object's `__dict__`. If it doesn't, it checks the class's `__dict__`, and if necessary, proceeds to base classes. If a descriptor is present, the descriptor's `__get__` or `__set__` methods are called.

Chapter 39: Metaclasses

Metaclasses are a powerful and advanced feature in Python that allows you to control the creation of classes. In Python, classes themselves are instances of metaclasses. Essentially, a metaclass is a class for classes—it's the "blueprint" for creating classes, just like classes are blueprints for creating objects.

What is a Metaclass?

A metaclass in Python is a class that defines the behavior of other classes. When you define a class, Python uses a metaclass to create the class itself. By default, the metaclass for all classes in Python is `type`, but you can customize this behavior by defining your own metaclasses.

Section 39.1: Basic Metaclasses

In this section, we will look at the basics of how to define and use metaclasses.

Creating a Metaclass

A metaclass is defined by subclassing `type`. The metaclass can be used to customize class creation, attribute management, and other behaviors. To use a custom metaclass, you assign it to the `metaclass` attribute of a class.

Example:

```
python
Copy code
# Defining a metaclass
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name} with metaclass {cls._
_name__}")
        return super().__new__(cls, name, bases, dct)

# Creating a class using the metaclass
class MyClass(metaclass=MyMeta):
    pass

# Output:
# Creating class MyClass with metaclass MyMeta
```

In this example:

- `MyMeta` is a metaclass that inherits from `type`.
- The `__new__` method in `MyMeta` is overridden to control the creation of the class.

- `MyClass` uses `MyMeta` as its metaclass, so when `MyClass` is created, the `__new__` method of `MyMeta` is called.

Section 39.2: Singletons using Metaclasses

One common use case for metaclasses is implementing the Singleton pattern, which ensures that a class has only one instance.

Singleton with Metaclasses

Here's how you can implement the Singleton pattern using a metaclass:

Example:

```
python
Copy code
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            # Create the instance only if it doesn't alread
y exist
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]

# Using the Singleton metaclass
class SingletonClass(metaclass=SingletonMeta):
    pass

# Create two instances of SingletonClass
instance1 = SingletonClass()
instance2 = SingletonClass()

# Check if both instances are the same
print(instance1 is instance2)  # True
```

Output:

```
graphql
Copy code
True
```

In this example:

- `SingletonMeta` is a metaclass that ensures only one instance of a class is created by checking if the class already has an instance.
- When a new instance of `SingletonClass` is created, the metaclass controls the instantiation, ensuring that only one instance is returned.

Section 39.3: Using a Metaclass

In this section, we explore how to use a metaclass to modify class behaviors, including validation, customization, and adding new attributes or methods to a class.

Example: Adding Methods to Classes with Metaclasses

```
python
Copy code
class AddMethodMeta(type):
    def __new__(cls, name, bases, dct):
        # Add a new method to the class
        def new_method(self):
            return f"{self.__class__.__name__} method added
by metaclass"

        dct['new_method'] = new_method
        return super().__new__(cls, name, bases, dct)

# Create a class using the metaclass
class MyClass(metaclass=AddMethodMeta):
    pass

# Creating an instance of MyClass
obj = MyClass()
```

```
# Calling the method added by the metaclass
print(obj.new_method())  # Output: MyClass method added by
metaclass
```

In this example:

- `AddMethodMeta` is a metaclass that dynamically adds a method `new_method` to the class it creates.
- The `new_method` is added to `MyClass` when it is created, and it is available to any instances of the class.

Section 39.4: Introduction to Metaclasses

Metaclasses provide a way to control how classes are created and how they behave. This section introduces metaclasses in general, explaining how Python's class system works and how you can customize class creation by using metaclasses.

When to Use Metaclasses

Metaclasses are generally used in advanced scenarios, such as:

- Creating Singleton classes.
- Validating class definitions or enforcing coding standards.
- Adding methods or attributes dynamically to classes.
- Implementing domain-specific languages (DSLs) or other design patterns.

Section 39.5: Custom Functionality with Metaclasses

In this section, we explore how to implement custom functionality using metaclasses, such as validating class attributes or modifying their behavior when they are defined.

Example: Validating Class Attributes with Metaclasses

```
python
Copy code
class AttributeValidatorMeta(type):
```

```
    def __new__(cls, name, bases, dct):
        # Ensure the class has a 'name' attribute
        if 'name' not in dct:
            raise TypeError("Class must have a 'name' attri
bute")
        return super().__new__(cls, name, bases, dct)

# Valid class
class ValidClass(metaclass=AttributeValidatorMeta):
    name = "Valid"

# Invalid class (raises TypeError)
try:
    class InvalidClass(metaclass=AttributeValidatorMeta):
        pass
except TypeError as e:
    print(e)  # Output: Class must have a 'name' attribute
```

Explanation:

- The `AttributeValidatorMeta` metaclass checks if a class being created has a `name` attribute.
- If the `name` attribute is missing, it raises a `TypeError`.

Section 39.6: The Default Metaclass

In Python, the default metaclass for all classes is `type`. The `type` metaclass is responsible for the creation of new classes. It is possible to use `type` directly as a metaclass or subclass it to create custom metaclasses.

Using `type` as a Metaclass

```
python
Copy code
# Using 'type' as the metaclass directly
class MyClass(metaclass=type):
    pass
```

```
# Checking the type of MyClass
print(type(MyClass))  # <class 'type'>
```

In this example, `type` is explicitly used as the metaclass, and it behaves as expected.

Chapter 40: String Formatting

String formatting in Python allows you to create formatted strings by embedding expressions or variables inside the string. It provides a clean way to construct strings, especially when dealing with variables and complex expressions.

Section 40.1: Basics of String Formatting

In Python, there are multiple ways to format strings, and the most common methods are:

1. **Using `%` operator** (older style, now less common):
    - This method uses placeholders in a string and the values to be inserted are provided in a tuple.

```
python
Copy code
name = "ManoharJoshi"
age = 25
formatted_string = "Name: %s, Age: %d" % (name, age)
print(formatted_string)
```

Output:

```
yaml
Copy code
```

```
Name: ManoharJoshi, Age: 25
```

2. **Using `.format()` method** (introduced in Python 2.7):
   - This method is more flexible and allows you to insert values into placeholders `{}`.

```
python
Copy code
name = "ManoharJoshi"
age = 25
formatted_string = "Name: {}, Age: {}".format(name, age)
print(formatted_string)
```

Output:

```
yaml
Copy code
Name: ManoharJoshi, Age: 25
```

3. **Using f-strings (formatted string literals)** (introduced in Python 3.6):
   - This method is the most modern and convenient way, where variables are directly embedded within the string.

```
python
Copy code
name = "ManoharJoshi"
age = 25
formatted_string = f"Name: {name}, Age: {age}"
print(formatted_string)
```

Output:

```
yaml
Copy code
Name: ManoharJoshi, Age: 25
```

Section 40.2: Alignment and Padding

You can use string formatting to align text and pad it with spaces or other characters.

Example of Padding and Alignment:

```
python
Copy code
# Left-align with padding
print(f"{'HiringHustle':<20}")  # Left-aligned, padded with
spaces
# Right-align with padding
print(f"{'HiringHustle':>20}")  # Right-aligned, padded wit
h spaces
# Center-align with padding
print(f"{'HiringHustle':^20}")  # Center-aligned, padded wi
th spaces
```

Output:

```
markdown
Copy code
HiringHustle
         HiringHustle
   HiringHustle
```

Using Custom Padding Character:

```
python
Copy code
# Padding with a different character
print(f"{'HiringHustle':*^20}")  # Center-aligned, padded w
ith asterisks
```

Output:

```
markdown
Copy code
****HiringHustle****
```

Section 40.3: Format Literals (f-string)

F-strings (formatted string literals) are a powerful feature for formatting strings in Python 3.6 and later. They allow you to embed expressions inside string literals using curly braces `{}` and prefix the string with `f`.

Example:

```
python
Copy code
name = "RishiKumar"
age = 30
formatted_string = f"Name: {name}, Age: {age}"
print(formatted_string)
```

Output:

```
yaml
Copy code
Name: RishiKumar, Age: 30
```

F-strings can also evaluate expressions inside the curly braces:

```
python
Copy code
x = 5
formatted_string = f"The square of {x} is {x**2}"
print(formatted_string)
```

Output:

```
csharp
Copy code
The square of 5 is 25
```

Section 40.4: Float Formatting

Python allows you to format floating-point numbers in specific ways, such as limiting the number of decimal places or using scientific notation.

Example of Float Formatting:

```
python
Copy code
pi = 3.141592653589793
print(f"{pi:.2f}")  # Round to 2 decimal places
print(f"{pi:.3e}")  # Scientific notation with 3 decimals
```

Output:

```
Copy code
3.14
3.142e+00
```

In this case, `.2f` formats the floating-point number to 2 decimal places, and `.3e` formats the number in scientific notation with 3 decimals.

Section 40.5: Named Placeholders

Instead of relying on positional arguments in the `.format()` method or f-strings, you can use named placeholders to make the format clearer.

Example:

```
python
Copy code
formatted_string = "Name: {name}, Age: {age}".format(name
="ManoharJoshi", age=25)
print(formatted_string)
```

Output:

```
yaml
Copy code
Name: ManoharJoshi, Age: 25
```

This method is especially useful when you have many variables to insert into the string.

Section 40.6: String Formatting with Datetime

Python provides a rich set of formatting options for date and time, which can be accessed through the `strftime` method.

Example with datetime:

```
python
Copy code
from datetime import datetime

current_time = datetime.now()
formatted_time = current_time.strftime("Today is %B %d, %Y
and the time is %H:%M:%S")
```

```
print(formatted_time)
```

Output:

```
csharp
Copy code
Today is December 29, 2024 and the time is 14:45:30
```

You can format the date and time with different codes:

- `%B` for the full month name.
- `%d` for the day of the month.
- `%Y` for the year.
- `%H` for the hour (24-hour clock).
- `%M` for the minute.
- `%S` for the second.

Section 40.7: Formatting Numerical Values

You can format large numbers with commas or in fixed-width forms, or even show the number in hexadecimal, binary, or octal forms.

Example:

```
python
Copy code
large_number = 1000000
print(f"{large_number:,}")  # Adds commas to large numbers
```

Output:

```
Copy code
1,000,000
```

You can also format numbers in binary, octal, or hexadecimal:

```
python
Copy code
number = 255
print(f"{number:b}")  # Binary
print(f"{number:o}")  # Octal
print(f"{number:x}")  # Hexadecimal
```

Output:

```
Copy code
11111111
377
ff
```

Section 40.8: Nested Formatting

You can nest formatting expressions inside f-strings for more complex formatting.

Example:

```
python
Copy code
value = 5
formatted_string = f"The value is {'{0:>5}'.format(value)}"
print(formatted_string)
```

Output:

```
csharp
Copy code
The value is     5
```

Section 40.9: Format using `getitem` and `getattr`

You can use `getitem` and `getattr` to format attributes or elements of an object dynamically.

Example:

```
python
Copy code
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("ManoharJoshi", 25)
formatted_string = f"Name: {p['name']}, Age: {p['age']}"
print(formatted_string)
```

This example uses `getitem` and `getattr` to access the attributes dynamically for formatting.

Section 40.10: Padding and Truncating Strings, Combined

You can combine padding and truncating to control how strings are displayed.

Example:

```
python
Copy code
# Truncating a string and padding it
long_string = "This is a very long string"
formatted_string = f"{long_string:.10}..."
print(formatted_string)
```

Output:

```
csharp
Copy code
This is a ...
```

In this example, `.10` truncates the string to 10 characters, and `...` is added as the suffix.

Section 40.11: Custom Formatting for a Class

You can define custom formatting behavior for a class by overriding the `__format__` method. This allows you to customize how instances of a class are formatted using the `format()` function or f-strings.

Example:

```
python
Copy code
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __format__(self, format_spec):
        return f"Name: {self.name}, Age: {self.age}"

p = Person("RishiKumar", 30)
formatted_string = f"{p}"
print(formatted_string)
```

Output:

```
yaml
Copy code
Name: RishiKumar, Age: 30
```

Chapter 41: String Methods

Python provides a wealth of built-in methods that allow you to manipulate and work with strings efficiently. These methods help with common tasks such as altering string capitalization, replacing substrings, splitting strings, and more.

Section 41.1: Changing the Capitalization of a String

You can change the capitalization of a string using various methods:

- `upper()`: Converts all characters to uppercase.
- `lower()`: Converts all characters to lowercase.
- `capitalize()`: Capitalizes the first letter of the string.
- `title()`: Capitalizes the first letter of each word.
- `swapcase()`: Swaps case for all characters.

Examples:

```
python
Copy code
text = "HiringHustle"

print(text.upper())         # "HIRINGHUSTLE"
print(text.lower())         # "hiringhustle"
print(text.capitalize())     # "Hiringhustle"
print(text.title())         # "Hiringhustle"
print(text.swapcase())      # "hIRINGhUSTLE"
```

Section 41.2: `str.translate`: Translating Characters in a String

The `str.translate()` method allows you to map each character in a string to another character using a translation table.

Example:

```
python
Copy code
# Create a translation table
trans_table = str.maketrans('Hh', 'Xx')

text = "HiringHustle"
translated_text = text.translate(trans_table)
print(translated_text)  # "XiringXustle"
```

In this example, all occurrences of 'H' are replaced with 'X', and 'h' with 'x'.

Section 41.3: `str.format` and f-strings: Format Values into a String

Both `.format()` and f-strings allow you to insert variables into a string, making the string more readable and dynamic.

Example using `str.format`:

```
python
Copy code
name = "ManoharJoshi"
age = 30
formatted_string = "Name: {}, Age: {}".format(name, age)
print(formatted_string)
```

Example using f-strings:

```
python
Copy code
formatted_string = f"Name: {name}, Age: {age}"
print(formatted_string)
```

Both methods will output:

```
yaml
Copy code
Name: ManoharJoshi, Age: 30
```

Section 41.4: String Module's Useful Constants

The `string` module provides constants like `string.ascii_letters`, `string.digits`, and others to easily work with strings.

Example:

```
python
Copy code
import string

print(string.ascii_letters)   # 'abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print(string.digits)          # '0123456789'
print(string.punctuation)     # '!"#$%&\'()*+,-./:;<=>?@
[\\]^_`{|}~'
```

Section 41.5: Stripping Unwanted Leading/Trailing Characters from a String

The `strip()`, `lstrip()`, and `rstrip()` methods are used to remove unwanted characters (by default whitespace) from the beginning and/or end of a string.

Examples:

```
python
Copy code
text = "  HiringHustle  "

print(text.strip())   # "HiringHustle"
print(text.lstrip())  # "HiringHustle  "
```

```
print(text.rstrip())  # "  HiringHustle"
```

You can also specify which characters to remove:

```
python
Copy code
text = "xxHiringHustlexx"
print(text.strip('x'))  # "HiringHustle"
```

Section 41.6: Reversing a String

To reverse a string, you can use slicing with a step of `-1`.

Example:

```
python
Copy code
text = "HiringHustle"
reversed_text = text[::-1]
print(reversed_text)  # "elt suHgniriH"
```

Section 41.7: Split a String Based on a Delimiter into a List of Strings

The `split()` method splits a string into a list based on a delimiter (by default, whitespace).

Example:

```
python
Copy code
text = "HiringHustle ManoharJoshi RishiKumar"
split_text = text.split()
print(split_text)  # ['HiringHustle', 'ManoharJoshi', 'Rish
```

```
iKumar']
```

You can also specify a delimiter:

```
python
Copy code
text = "apple,banana,orange"
split_text = text.split(',')
print(split_text)  # ['apple', 'banana', 'orange']
```

Section 41.8: Replace All Occurrences of One Substring with Another Substring

The `replace()` method allows you to replace all occurrences of a substring with another substring.

Example:

```
python
Copy code
text = "HiringHustle is great"
new_text = text.replace("HiringHustle", "JobHunt")
print(new_text)  # "JobHunt is great"
```

Section 41.9: Testing What a String is Composed Of

You can test whether a string is composed of certain types of characters using methods like `isalpha()`, `isdigit()`, and `isspace()`.

Examples:

```
python
Copy code
text = "HiringHustle"
print(text.isalpha())   # True, all characters are letters
```

```
number = "12345"
print(number.isdigit()) # True, all characters are digits

whitespace = "   "
print(whitespace.isspace())  # True, all characters are spa
ces
```

Section 41.10: String Contains

You can check if a string contains a certain substring using the `in` keyword.

Example:

```
python
Copy code
text = "HiringHustle"
print("Hustle" in text)  # True
print("Job" in text)     # False
```

Section 41.11: Join a List of Strings into One String

The `join()` method allows you to join a list of strings into a single string with a specified separator.

Example:

```
python
Copy code
words = ["HiringHustle", "ManoharJoshi", "RishiKumar"]
joined_text = " ".join(words)
print(joined_text)  # "HiringHustle ManoharJoshi RishiKuma
r"
```

You can use any separator, like commas or dashes:

```
python
Copy code
joined_text = ", ".join(words)
print(joined_text)  # "HiringHustle, ManoharJoshi, RishiKum
ar"
```

Section 41.12: Counting Number of Times a Substring Appears in a String

The `count()` method counts how many times a substring appears in a string.

Example:

```
python
Copy code
text = "HiringHustle, HiringHustle, HiringHustle"
print(text.count("HiringHustle"))  # 3
```

Section 41.13: Case Insensitive String Comparisons

You can compare strings case-insensitively using `lower()` or `upper()` to normalize the case before comparison.

Example:

```
python
Copy code
text1 = "HiringHustle"
text2 = "hiringhustle"
print(text1.lower() == text2.lower())  # True
```

Section 41.14: Justify Strings

The `ljust()`, `rjust()`, and `center()` methods are used to pad strings and align them.

Examples:

```
python
Copy code
text = "HiringHustle"
print(text.ljust(20, '-'))  # "HiringHustle------"
print(text.rjust(20, '-'))  # "------HiringHustle"
print(text.center(20, '-')) # "--HiringHustle---"
```

Section 41.15: Test the Starting and Ending Characters of a String

The `startswith()` and `endswith()` methods allow you to test whether a string starts or ends with a specific substring.

Examples:

```
python
Copy code
text = "HiringHustle"
print(text.startswith("Hiring"))  # True
print(text.endswith("Hustle"))    # True
```

Section 41.16: Conversion Between `str` or `bytes` Data and Unicode Characters

You can encode a string into bytes and decode it back to a string using the `encode()` and `decode()` methods.

Example:

```
python
Copy code
```

```
text = "HiringHustle"
encoded_text = text.encode('utf-8')  # Converts to bytes
decoded_text = encoded_text.decode('utf-8')  # Converts bac
k to string
print(encoded_text)  # b'HiringHustle'
print(decoded_text)  # HiringHustle
```

Chapter 42: Using Loops within Functions

Loops are powerful tools when incorporated within functions, enabling repetitive operations without duplicating code. This chapter covers how to use loops in combination with functions.

Section 42.1: Return Statement Inside a Loop in a Function

Using a `return` statement inside a loop allows you to exit the function as soon as a certain condition is met, which is helpful when you want to terminate execution early and return a result based on a condition.

Example:

```
python
Copy code
def find_first_even(numbers):
    for number in numbers:
        if number % 2 == 0:
            return number  # Returns the first even number
    return None  # Returns None if no even number is found

numbers = [1, 3, 7, 8, 11]
result = find_first_even(numbers)
print(result)  # 8
```

In this example, the function returns the first even number it finds and exits the loop early.

If no even number is found, the function returns `None`.

Chapter 43: Importing Modules

In Python, modules are an essential part of writing reusable and modular code. This chapter discusses the various ways to import modules and the rules governing imports.

Section 43.1: Importing a Module

The `import` statement allows you to bring external Python files (modules) into your program. There are several ways to import modules depending on the desired functionality.

Example:

```
python
Copy code
import math  # Import the entire math module
print(math.sqrt(16))  # Output: 4.0
```

You can also give the module a shorter alias using `as`:

```
python
Copy code
import math as m
print(m.sqrt(16))  # Output: 4.0
```

Section 43.2: The `__all__` Special Variable

The `__all__` variable in a module controls what is imported when `from module import *` is used. If `__all__` is not defined, all top-level names are imported by default.

Example:

```
python
Copy code
# In a module (e.g., mymodule.py)
__all__ = ['function_a', 'function_b']

def function_a():
    pass

def function_b():
    pass

def function_c():  # Not imported by default
    pass
```

Now, when you import everything from the module, only `function_a` and `function_b` are imported:

```
python
Copy code
from mymodule import *
```

Section 43.3: Import Modules from an Arbitrary Filesystem Location

Sometimes, you need to import modules that aren't in the default Python search path. You can use `sys.path` to add directories to the module search path.

Example:

```
python
Copy code
import sys
sys.path.append('/path/to/your/module')

import yourmodule  # Now you can import the module from the
```

```
custom path
```

Section 43.4: Importing All Names from a Module

The `from module import *` syntax allows you to import all public names from a module. However, it is not recommended because it can lead to namespace pollution.

Example:

```
python
Copy code
from math import *
print(sqrt(16))  # No need to reference 'math' here
```

Section 43.5: Programmatic Importing

You can import modules dynamically using the `__import__()` function or `importlib`.

Example using `__import__`:

```
python
Copy code
module_name = "math"
math_module = __import__(module_name)
print(math_module.sqrt(16))  # Output: 4.0
```

This can be useful when the module name is determined dynamically at runtime.

Section 43.6: PEP8 Rules for Imports

PEP8, the Python style guide, recommends that imports be organized in a specific way:

1. **Standard library imports** first.
2. **Related third-party imports** next.
3. **Local application or library imports** last.
4. You should also group imports with one line between each group.

Example:

```
python
Copy code
import os
import sys

import requests

from mymodule import myfunction
```

Section 43.7: Importing Specific Names from a Module

Instead of importing the entire module, you can import specific functions or variables from a module.

Example:

```
python
Copy code
from math import sqrt, pi
print(sqrt(16))  # 4.0
print(pi)        # 3.141592653589793
```

Section 43.8: Importing Submodules

Many libraries are organized into submodules, and you can import them using dot notation.

Example:

```
python
Copy code
import os.path
print(os.path.join("folder", "file.txt"))  # "folder/file.t
xt"
```

Section 43.9: Re-importing a Module

If you modify a module and want to reload it without restarting your Python session, you can use `importlib.reload()`.

Example:

```
python
Copy code
import importlib
import mymodule

importlib.reload(mymodule)  # Reload the module
```

Section 43.10: `__import__()` Function

The `__import__()` function is a low-level function used for importing modules dynamically. You can use it to programmatically load a module by name.

Example:

```
python
Copy code
module_name = "math"
math_module = __import__(module_name)
print(math_module.sqrt(16))  # Output: 4.0
```

Chapter 44: Difference Between Module and Package

In Python, both modules and packages are essential for organizing code, but they serve different purposes. Understanding the difference between them is important for managing larger Python projects.

Section 44.1: Modules

A **module** is a single Python file that contains code such as functions, classes, or variables. It can be imported and used in other Python scripts or modules.

Example:

```
python
Copy code
# math_operations.py (Module)
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

To use this module in another Python file, you simply import it:

```
python
Copy code
import math_operations
result = math_operations.add(5, 3)
print(result)  # Output: 8
```

Section 44.2: Packages

A **package** is a collection of modules grouped together in a directory hierarchy. A package contains a special file named `__init__.py`, which tells Python that the directory should be treated as a package.

Example:

Consider the following directory structure:

```
markdown
Copy code
math_package/
    __init__.py
    operations.py
    geometry.py
```

Here, `math_package` is a package, and `operations.py` and `geometry.py` are modules inside it. To import a module from the package:

```
python
Copy code
from math_package import operations
result = operations.add(5, 3)
print(result)  # Output: 8
```

A package allows you to organize multiple related modules into a single namespace.

Chapter 45: Math Module

Python's `math` module provides mathematical functions and constants. This chapter covers various functions in the `math` module that help in mathematical computations.

Section 45.1: Rounding: round, floor, ceil, trunc

The `math` module offers several functions to round numbers and perform floor, ceiling, and truncation operations.

- `round()`: Rounds a number to the nearest integer.

  ```
  python
  Copy code
  round(3.5)  # 4
  round(3.4)  # 3
  ```

- `math.floor()`: Returns the largest integer less than or equal to a given number.

```
python
Copy code
import math
math.floor(3.7)  # 3
```

- `math.ceil()`: Returns the smallest integer greater than or equal to a given number.

```
python
Copy code
import math
math.ceil(3.1)  # 4
```

- `math.trunc()`: Truncates a number by removing its decimal part.

```
python
Copy code
import math
math.trunc(3.7)  # 3
```

Section 45.2: Trigonometry

The `math` module provides several trigonometric functions for working with angles (in radians).

- `math.sin(x)`: Returns the sine of `x`.
- `math.cos(x)`: Returns the cosine of `x`.
- `math.tan(x)`: Returns the tangent of `x`.
- `math.asin(x)`: Returns the arcsine (inverse of sine) of `x`.
- `math.acos(x)`: Returns the arccosine (inverse of cosine) of `x`.
- `math.atan(x)`: Returns the arctangent (inverse of tangent) of `x`.

```
python
Copy code
import math
angle = math.radians(45)  # Convert angle to radians
print(math.sin(angle))  # Output: 0.7071067811865475
```

Section 45.3: Pow for Faster Exponentiation

The `math.pow()` function is used for exponentiation.

```
python
Copy code
import math
print(math.pow(2, 3))  # Output: 8.0 (2 raised to the power
of 3)
```

Alternatively, the `**` operator can be used in Python for exponentiation.

```
python
Copy code
print(2 ** 3)  # Output: 8
```

Section 45.4: Infinity and NaN ("Not a Number")

The `math` module includes constants for **infinity** and **NaN (Not a Number)**.

- `math.inf`: Positive infinity.
- `math.inf`: Negative infinity.
- `math.nan`: Not a number.

```
python
Copy code
import math
print(math.inf)  # Output: inf
```

```
print(math.nan)  # Output: nan
```

Section 45.5: Logarithms

The `math` module provides functions for logarithmic calculations.

- `math.log(x, base)`: Returns the logarithm of `x` to the given `base`.
- `math.log10(x)`: Returns the base-10 logarithm of `x`.
- `math.log2(x)`: Returns the base-2 logarithm of `x`.

```
python
Copy code
import math
print(math.log(100, 10))  # Output: 2.0 (logarithm base 10
of 100)
print(math.log10(100))  # Output: 2.0
```

Section 45.6: Constants

The `math` module provides some important mathematical constants:

- `math.pi`: The mathematical constant pi (π).
- `math.e`: The mathematical constant e (Euler's number).

```
python
Copy code
import math
print(math.pi)  # Output: 3.141592653589793
print(math.e)   # Output: 2.718281828459045
```

Section 45.7: Imaginary Numbers

The `cmath` module (for complex numbers) provides functions to deal with imaginary numbers. You can convert numbers to complex type or perform complex arithmetic using `cmath`.

```
python
Copy code
import cmath
z = cmath.sqrt(-1)
print(z)  # Output: 1j
```

Section 45.8: Copying Signs

You can copy the sign of one number to another using `math.copysign(x, y)`.

```
python
Copy code
import math
print(math.copysign(3, -2))  # Output: -3.0 (copy sign of -
2 to 3)
```

Section 45.9: Complex Numbers and the `cmath` Module

The `cmath` module offers functions for complex numbers and can handle operations involving imaginary numbers.

- `cmath.phase(z)`: Returns the phase (or angle) of a complex number `z`.
- `cmath.polar(z)`: Converts a complex number to polar coordinates.
- `cmath.rect(r, phi)`: Converts polar coordinates to rectangular form.

```
python
Copy code
import cmath
z = complex(3, 4)  # Complex number
print(cmath.polar(z))  # Output: (5.0, 0.9272952180016122)
(magnitude and phase)
```

Chapter 46: Complex Math

Python supports complex numbers, and this chapter explores more advanced and basic operations on complex numbers.

Section 46.1: Advanced Complex Arithmetic

Python's support for complex numbers allows you to perform operations such as addition, subtraction, multiplication, and division with complex numbers. You can also use functions from the `cmath` module for advanced mathematical functions on complex numbers.

Example:

```
python
Copy code
import cmath

# Define complex numbers
z1 = complex(3, 4)  # 3 + 4j
z2 = complex(1, 2)  # 1 + 2j

# Complex arithmetic operations
sum_z = z1 + z2  # (3 + 4j) + (1 + 2j) = (4 + 6j)
diff_z = z1 - z2  # (3 + 4j) - (1 + 2j) = (2 + 2j)
product_z = z1 * z2  # (3 + 4j) * (1 + 2j) = (3 + 6j + 4j +
8j^2) = (3 + 10j - 8) = (-5 + 10j)
quotient_z = z1 / z2  # (3 + 4j) / (1 + 2j) = (11 + 2j) / 5
= (2.2 + 0.4j)

print("Sum:", sum_z)
print("Difference:", diff_z)
print("Product:", product_z)
print("Quotient:", quotient_z)
```

Section 46.2: Basic Complex Arithmetic

In Python, complex numbers are represented with a real and imaginary part using the `complex()` function or by using the suffix `j`.

Basic Operations:

- **Addition**: `z1 + z2`
- **Subtraction**: `z1 - z2`
- **Multiplication**: `z1 * z2`
- **Division**: `z1 / z2`

Example:

```
python
Copy code
# Using complex numbers directly
z1 = 3 + 4j
z2 = 1 + 2j

# Arithmetic operations
addition = z1 + z2  # Output: (4 + 6j)
subtraction = z1 - z2  # Output: (2 + 2j)

print("Addition:", addition)
print("Subtraction:", subtraction)
```

Chapter 47: Collections Module

Python's `collections` module provides alternatives to Python's general-purpose built-in types like lists and dictionaries. It includes specialized container datatypes.

Section 47.1: `collections.Counter`

The `Counter` class is a subclass of `dict` designed to count hashable objects. It helps in counting elements in an iterable or mapping.

Example:

```
python
Copy code
```

```
from collections import Counter

# Count the occurrences of elements in a list
counter = Counter([1, 2, 2, 3, 3, 3, 4])
print(counter)  # Output: Counter({3: 3, 2: 2, 1: 1, 4: 1})

# Count characters in a string
char_counter = Counter("hello")
print(char_counter)  # Output: Counter({'l': 2, 'h': 1,
'e': 1, 'o': 1})
```

Section 47.2: `collections.OrderedDict`

An `OrderedDict` is a subclass of `dict` that maintains the order of keys as they are added. This is useful when you need to preserve the order in which items were inserted into a dictionary.

Example:

```
python
Copy code
from collections import OrderedDict

# OrderedDict keeps the order of insertion
ordered_dict = OrderedDict()
ordered_dict['a'] = 1
ordered_dict['b'] = 2
ordered_dict['c'] = 3
print(ordered_dict)  # Output: OrderedDict([('a', 1), ('b',
2), ('c', 3)])
```

Section 47.3: `collections.defaultdict`

The `defaultdict` is a subclass of `dict` that provides a default value for missing keys.

Example:

```
python
Copy code
from collections import defaultdict

# Using defaultdict with a default factory function
d = defaultdict(int)  # Default value is 0
d['apple'] += 1
d['banana'] += 2
print(d)  # Output: defaultdict(<class 'int'>, {'apple': 1,
'banana': 2})
```

Section 47.4: `collections.namedtuple`

The `namedtuple` function is used to create a tuple subclass with named fields. It provides an easy way to define simple classes for storing data.

Example:

```
python
Copy code
from collections import namedtuple

# Define a namedtuple called Point with x and y fields
Point = namedtuple('Point', ['x', 'y'])

# Create an instance of Point
pt = Point(3, 4)
print(pt.x, pt.y)  # Output: 3 4
```

Section 47.5: `collections.deque`

A `deque` (double-ended queue) is a list-like container with fast appends and pops from both ends.

Example:

```
python
Copy code
from collections import deque

# Create a deque
dq = deque([1, 2, 3, 4])
dq.append(5)  # Add to the right
dq.appendleft(0)  # Add to the left
print(dq)  # Output: deque([0, 1, 2, 3, 4, 5])
```

Section 47.6: `collections.ChainMap`

A `ChainMap` groups multiple dictionaries or mappings together to create a single, updateable view.

Example:

```
python
Copy code
from collections import ChainMap

# Create two dictionaries
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}

# Create a ChainMap from the two dictionaries
chain_map = ChainMap(dict1, dict2)
print(chain_map['b'])  # Output: 2 (from dict1)
print(chain_map['c'])  # Output: 4 (from dict2)
```

Chapter 48: Operator Module

The `operator` module exports a set of functions that correspond to standard operators in Python. These can be useful for functional programming.

Section 48.1: `itemgetter`

`itemgetter` creates a callable object that can fetch an item from a sequence or mapping.

Example:

```
python
Copy code
from operator import itemgetter

# Use itemgetter to retrieve the second element from a tupl
e
data = [(1, 'a'), (2, 'b'), (3, 'c')]
getter = itemgetter(1)
print(getter(data[0]))  # Output: 'a'
```

Section 48.2: Operators as Alternative to an Infix Operator

Python's operator functions can be used to replace infix operators with functions that can be passed as arguments.

Example:

```
python
Copy code
from operator import add, mul

# Using operator functions
result_add = add(2, 3)  # 5
result_mul = mul(2, 3)  # 6
```

Section 48.3: `methodcaller`

The `methodcaller` function returns a callable that invokes a method on an object.

Example:

```
python
Copy code
from operator import methodcaller

# Use methodcaller to call 'lower' method on a string
to_lower = methodcaller('lower')
print(to_lower('HELLO'))  # Output: 'hello'
```

Chapter 49: JSON Module

The `json` module in Python is used for working with JSON (JavaScript Object Notation) data. JSON is a lightweight data interchange format that is easy to read and write for humans and machines alike. It is often used for transmitting data in web applications.

Section 49.1: Storing Data in a File

You can store Python objects as JSON in a file using the `json.dump()` function. This is useful for saving configuration files, data, and other structures in a standard format.

Example:

```
python
Copy code
import json

# Example Python dictionary
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Open a file in write mode and store JSON
with open('data.json', 'w') as f:
    json.dump(data, f)
```

Section 49.2: Retrieving Data from a File

To read the JSON data back from a file, use the `json.load()` function. This converts the JSON data into Python objects (like dictionaries).

Example:

```
python
Copy code
import json

# Read the JSON file and convert it to a Python dictionary
with open('data.json', 'r') as f:
    data = json.load(f)

print(data)  # Output: {'name': 'John', 'age': 30, 'city':
'New York'}
```

Section 49.3: Formatting JSON Output

When outputting JSON, you can format it to make it easier to read. The `json.dump()` and `json.dumps()` functions have parameters like `indent` to improve the output's readability.

Example:

```
python
Copy code
import json

# Example Python dictionary
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Write the JSON data with indentation
with open('data.json', 'w') as f:
    json.dump(data, f, indent=4)
```

The resulting JSON file will be formatted like this:

```
json
Copy code
{
    "name": "John",
    "age": 30,
    "city": "New York"
}
```

Section 49.4: `load` vs `loads`, `dump` vs `dumps`

- `json.load()`: Reads a JSON file and converts it to a Python object.
- `json.loads()`: Reads a JSON string and converts it to a Python object.
- `json.dump()`: Writes a Python object to a file as JSON.
- `json.dumps()`: Converts a Python object into a JSON string.

Example:

```
python
Copy code
import json

# Using load() and dump()
with open('data.json', 'w') as f:
    json.dump({'name': 'John', 'age': 30}, f)

# Using loads() and dumps()
json_str = '{"name": "John", "age": 30}'
data = json.loads(json_str)  # Converts JSON string to Pyth
on object
json_str = json.dumps(data)  # Converts Python object to JS
ON string
```

Section 49.5: Calling `json.tool` from the Command Line to Pretty-Print JSON Output

The `json.tool` module can be used directly from the command line to pretty-print JSON data. It is especially useful for debugging or viewing large JSON files.

Example:

```
bash
Copy code
# Command to pretty-print a JSON file
python -m json.tool data.json
```

This command will display the contents of `data.json` in a formatted, human-readable way.

Section 49.6: JSON Encoding Custom Objects

When working with custom objects, you may need to define how they should be serialized into JSON. You can achieve this by overriding the `default` method in a custom encoder class.

Example:

```
python
Copy code
import json

class CustomEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, CustomClass):
            return {'name': obj.name}
        return super().default(obj)

class CustomClass:
    def __init__(self, name):
        self.name = name
```

```
# Example usage
obj = CustomClass('Test')

# Encode the custom object into JSON
json_str = json.dumps(obj, cls=CustomEncoder)
print(json_str)  # Output: {"name": "Test"}
```

Section 49.7: Creating JSON from Python Dict

You can create a JSON string from a Python dictionary using `json.dumps()`.

Example:

```
python
Copy code
import json

data = {'name': 'Alice', 'age': 25}
json_str = json.dumps(data)

print(json_str)  # Output: '{"name": "Alice", "age": 25}'
```

Section 49.8: Creating Python Dict from JSON

To convert a JSON string into a Python dictionary, use `json.loads()`.

Example:

```
python
Copy code
import json

json_str = '{"name": "Alice", "age": 25}'
data = json.loads(json_str)
```

```
print(data)  # Output: {'name': 'Alice', 'age': 25}
```

Chapter 50: Sqlite3 Module

The `sqlite3` module allows you to interact with SQLite databases. SQLite is a lightweight, disk-based database that doesn't require a separate server process.

Section 50.1: SQLite3 - No Separate Server Process Required

SQLite databases are self-contained and do not require a server process. You can interact with the database directly using the `sqlite3` module.

Example:

```
python
Copy code
import sqlite3

# Connect to an SQLite database (it will be created if it d
oesn't exist)
conn = sqlite3.connect('example.db')

# Create a table
conn.execute('''CREATE TABLE IF NOT EXISTS users (id INTEGE
R PRIMARY KEY, name TEXT)''')

# Insert some data
conn.execute("INSERT INTO users (name) VALUES ('Alice')")
conn.commit()

# Close the connection
conn.close()
```

Section 50.2: Getting Values from the Database and Error Handling

You can query data from the SQLite database using the `SELECT` statement. You can also handle errors with try-except blocks.

Example:

```
python
Copy code
import sqlite3

try:
    # Connect to the SQLite database
    conn = sqlite3.connect('example.db')

    # Retrieve values
    cursor = conn.execute("SELECT * FROM users")
    for row in cursor:
        print(f"ID: {row[0]}, Name: {row[1]}")

except sqlite3.Error as e:
    print("SQLite error:", e)

finally:
    # Close the connection
    if conn:
        conn.close()
```

This example retrieves data from the `users` table and handles errors in case of issues like database connectivity problems.