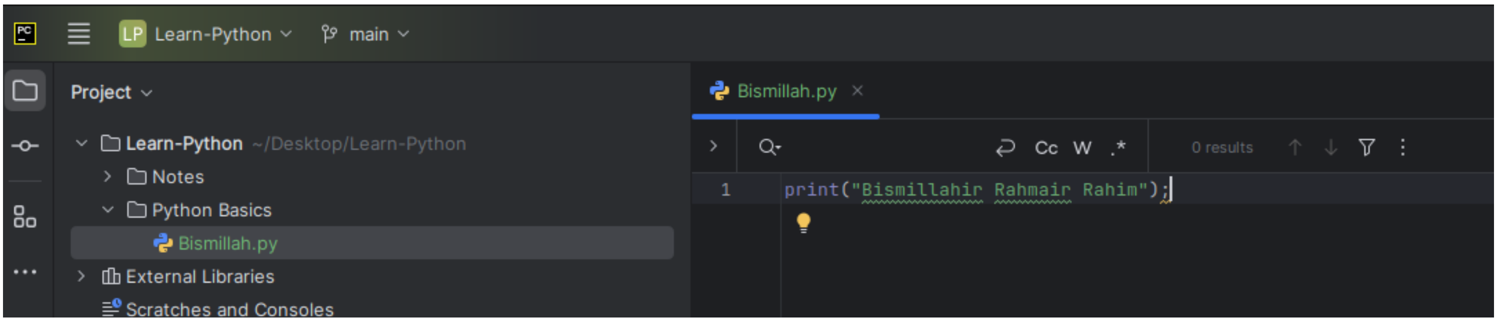
# Pycharm Basics

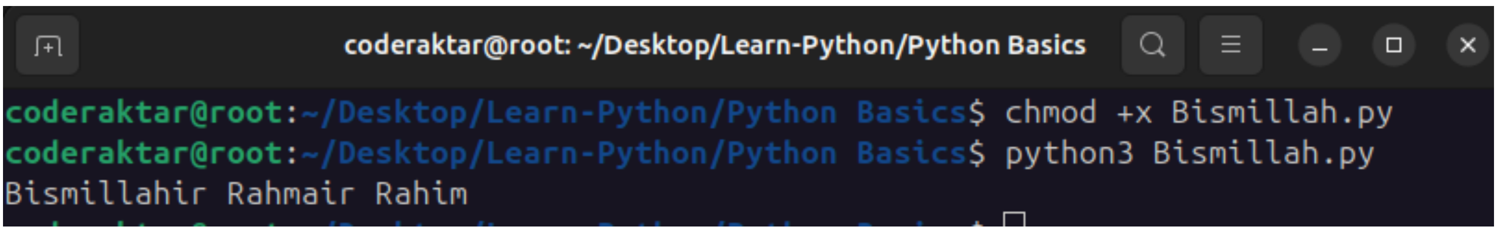
Duplicate Line: Ctrl + D
Search word in Current File: Ctrl + F
Search word in whole Project: Ctrl + shift + F
Search File in project: Shift double pressed

# Run Python From Command Line

Create File name Bismillah.py




Go to this Folder where file is located
1. Giving Execution Permission for write: chmod +x Bismillah.py
2. Run Python From Terminal: python3  Bismillah.py



# Python Basics

## Variable In Python

A **variable** in Python is used to store data that can be accessed and manipulated later in the code. Variables are created by assigning a value to a name using the `=` operator.

Here's a basic example:

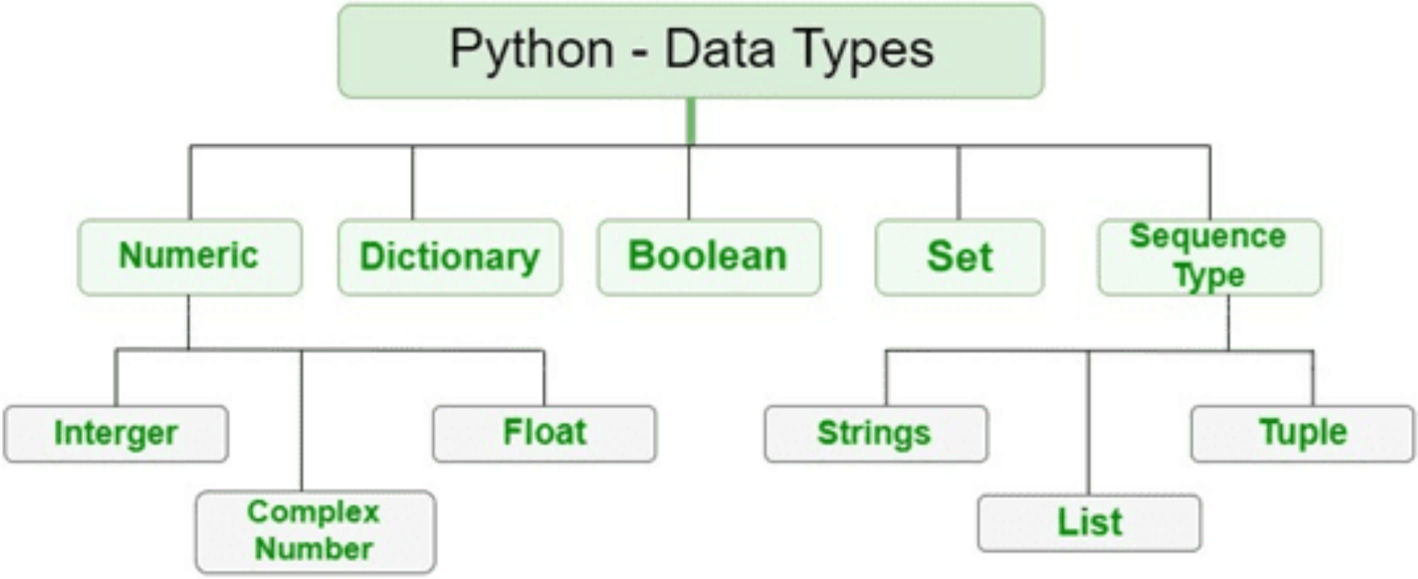
```
# Variable storing an integer
x = 5

# Variable storing a string
name = "Alice"

# Variable storing a floating-point number
pi = 3.14159

# Variable storing a list
```

numbers = [1, 2, 3, 4, 5]

### Key Points:
- **Dynamic Typing**: You don't need to declare the type of a variable. Python determines the type based on the value you assign.
- **Case Sensitivity**: Variables are case-sensitive (`x` and `X` are different).
- **Naming Rules**: Variable names must start with a letter or underscore (`_`), and can contain letters, numbers, and underscores. They can't start with a number or be a Python reserved keyword (like `for`, `while`, etc.).

Would you like to see more specific examples of variable use in Python?

# ***Built-in data type of python***




| Name | Type | Description |
|---|---|---|
| Integers | int | Whole numbers, such as: **3 300 200** |
| Floating point | float | Numbers with a decimal point: **2.3 4.6 100.0** |
| Strings | str | Ordered sequence of characters: **"hello" 'Sammy' "2000" "楽しい"** |
| Lists | list | Ordered sequence of objects: **[10,"hello",200.3]** |
| Dictionaries | dict | Unordered Key:Value pairs: **{"mykey" : "value" , "name" : "Frankie"}** |
| Tuples | tup | Ordered immutable sequence of objects: **(10,"hello",200.3)** |
| Sets | set | Unordered collection of unique objects: **{"a","b"}** |
| Booleans | bool | Logical value indicating **True** or **False** |

In Python, **everything is an object**, and that includes primitive types like `int`, `float`, `bool`, `str`, and even more complex structures like `list`, `tuple`, `set`, `dict`, and others. Let's break this down in detail:

## In Python, Everything is an Object

• Python is an object-oriented language, and everything you work with is an **instance (object)** of some **class**.
• Even basic types like integers, floats, booleans, and strings are instances of their respective classes.

For example:
◇ `int` is a class, and any integer value you use is an instance (object) of the `int` class.
◇ `float` is a class, and any floating-point number you use is an instance (object) of the `float` class.
◇ `bool` is a class, and `True` and `False` are objects of the `b-ool` class.

# *typecasting*

Typecasting in Python, also known as **type conversion**, refers to the process of converting one data type to another. Python provides both **implicit** and **explicit** typecasting mechanisms, allowing you to seamlessly work with different data types as needed.

Understanding typecasting is essential for:

• **Data Manipulation:** Converting data types to perform specific operations.
• **Data Validation:** Ensuring data is in the correct format before processing.
• **Interoperability:** Facilitating operations between different data types.

Typecasting in Python, also known as **type conversion**, refers to the process of converting one data type to another. Python provides both **implicit** and **explicit** typecasting mechanisms, allowing you to seamlessly work with different data types as needed.
Understanding typecasting is essential for:
• **Data Manipulation:** Converting data types to perform specific operations.
• **Data Validation:** Ensuring data is in the correct format before processing.
• **Interoperability:** Facilitating operations between different data types.

Let's delve into the details of typecasting in Python.

# 1. Implicit Typecasting

**Implicit typecasting** occurs when Python automatically converts one data type to another without any explicit instruction from the programmer. This typically happens in operations involving mixed data types to prevent data loss.

## 2. Explicit Typecasting

**Explicit typecasting** requires the programmer to manually convert data types using built-in functions. This is useful when automatic type conversion isn't possible or desired.

**Common Typecasting Functions:**

| Function | Purpose |
|---|---|
| `int()` | Converts to integer |
| `float()` | Converts to float |
| `str()` | Converts to string |
| `list()` | Converts to list |
| `tuple()` | Converts to tuple |
| `set()` | Converts to set |
| `dict()` | Converts to dictionary (from key-value pairs) |

## ***arithmetic operators***

| Operator | Name | Description | Example | Result |
|---|---|---|---|---|
| `+` | Addition | Adds two operands | `5 + 3` | `8` |
| `-` | Subtraction | Subtracts the second operand from the first | `5 - 3` | `2` |
| `*` | Multiplication | Multiplies two operands | `5 * 3` | `15` |
| `/` | Division | Divides the first operand by the second (always returns float) | `5 / 2` | `2.5` |
| `//` | Floor Division | Divides and returns the integer part of the quotient | `5 // 2` | `2` |
| `%` | Modulus (Remainder) | Returns the remainder after division | `5 % 2` | `1` |
| `**` | Exponentiation | Raises the first operand to the power of the second | `5 ** 2` | `25` |
| `()` | Parentheses | Used to change the precedence of operations | `(5 + 3) * 2` | `16` |

# Operator Precedence

When multiple operators are used in an expression, **operator precedence** determines the order in which the operations are performed. Here's the hierarchy (from highest to lowest):

1. **Parentheses** `()`
2. **Exponentiation** `**`
3. **Unary plus and minus** `+x`, `-x`
4. **Multiplication**, **Division**, **Floor Division**, **Modulus** `*`, `/`, `//`, `%`
5. **Addition** and **Subtraction** `+`, `-`

Example:

```
result = 3 + 4 * 2 ** 2 / (1 - 5) % 2
print(result)  # Output: 3.0
```

**Step-by-Step Evaluation:**

1. **Parentheses:** `(1 - 5)` → `-4`
2. **Exponentiation:** `2 ** 2` → `4`
3. **Multiplication:** `4 * 4` → `16`
4. **Division:** `16 / -4` → `-4.0`
5. **Modulus:** `-4.0 % 2` → `0.0` (since `-4.0` is evenly divisible by `2`)
6. **Addition:** `3 + 0.0` → `3.0`

# *assignment Operator*

## Basic Assignment Operator (`=`)

The **basic assignment operator** `=` is used to assign a value to a variable. It's the most fundamental way to store data in Python.

## Syntax:

```
variable_name = value
```

## Examples:

```
# Assigning an integer
x = 10
print(x)  # Output: 10

# Assigning a string
name = "Alice"
print(name)  # Output: Alice

# Assigning a float
pi = 3.1415
print(pi)  # Output: 3.1415

# Assigning a boolean
is_active = True
print(is_active)  # Output: True
```

## Key Points:

- **Right-Associative:** The assignment operator is right-associative, meaning the expression on the right is evaluated first, then assigned to the variable on the left.

```python
x = y = 5
print(x)  # Output: 5
print(y)  # Output: 5
```

## Chained Assignment

**Chained assignment** allows you to assign the same value to multiple variables in a single line. This can make your code more concise.

```python
a = b = c = 100
print(a)  # Output: 100
print(b)  # Output: 100
print(c)  # Output: 100

# Assigning the same list to multiple variables
list1 = list2 = []
list1.append(1)
print(list2)  # Output: [1]
```

## Table of Augmented Assignment Operators:

| Operator | Description | Equivalent To | Example |
|---|---|---|---|
| `+=` | Add and assign | `x = x + y` | `x += y` |
| `-=` | Subtract and assign | `x = x - y` | `x -= y` |
| `*=` | Multiply and assign | `x = x * y` | `x *= y` |
| `/=` | Divide and assign | `x = x / y` | `x /= y` |
| `//=` | Floor divide and assign | `x = x // y` | `x //= y` |
| `%=` | Modulus and assign | `x = x % y` | `x %= y` |
| `**=` | Exponentiate and assign | `x = x ** y` | `x **= y` |
| `&=` | Bitwise AND and assign | `x = x & y` | `x &= y` |
| ` | =` | Bitwise OR and assign | `x = x |
| `^=` | Bitwise XOR and assign | `x = x ^ y` | `x ^= y` |
| `<<=` | Left shift and assign | `x = x << y` | `x <<= y` |
| `>>=` | Right shift and assign | `x = x >> y` | `x >>= y` |

Bitwise Assignment Operators (&=, |=, ^=, <<=, >>=)

```
x = 5     # Binary: 0101
y = 3     # Binary: 0011

x &= y    # Binary AND: 0001
print(x)  # Output: 1

x = 5
x |= y    # Binary OR: 0111
print(x)  # Output: 7
```

```
x = 5
x ^= y     # Binary XOR: 0110
print(x)   # Output: 6

x = 5
x <<= 1    # Left shift: 1010 (10 in decimal)
print(x)   # Output: 10

x = 5
x >>= 1    # Right shift: 0010 (2 in decimal)
print(x)   # Output: 2
```

Unpacking Assignments

**Unpacking assignments** allow you to assign values from iterable objects (like tuples, lists, or dictionaries) to multiple variables simultaneously. This enhances code readability and efficiency.

## a. Tuple Unpacking

Assigning elements of a tuple to variables

```
# Example
point = (10, 20)
x, y = point
print(x)  # Output: 10
print(y)  # Output: 20
```

## b. List Unpacking

Assigning elements of a list to variables.

```
# Example
colors = ["red", "green", "blue"]
first, second, third = colors
print(first)   # Output: red
print(second)  # Output: green
print(third)   # Output: blue
```

## c. Dictionary Unpacking

Assigning keys or values of a dictionary to variables.

```
# Assigning keys
person = {"name": "Alice", "age": 30}
key1, key2 = person
print(key1)  # Output: name
print(key2)  # Output: age

# Assigning values using .values()
value1, value2 = person.values()
print(value1)  # Output: Alice
print(value2)  # Output: 30

# Assigning key-value pairs using .items()
(k1, v1), (k2, v2) = person.items()
```

print(k1, v1)  # Output: name Alice
print(k2, v2)  # Output: age 30

## Key Points:

• **Matching Structure:** The number of variables on the left must match the number of elements in the iterable.

# This will raise a ValueError
a, b = [1, 2, 3]

• Using Asterisk (*) for Variable-Length Unpacking:

numbers = [1, 2, 3, 4, 5]
first, *middle, last = numbers
print(first)   # Output: 1
print(middle)  # Output: [2, 3, 4]
print(last)    # Output: 5

# *Immutable and Mutable*

# *Object(Importand)*

## 1. Immutable Objects

• **Definition:** Immutable objects cannot be changed once they are created.
• **Example:**◇ **Integers (`int`)**: If you want to change the value of an integer, you must reassign it to a new integer. For example:

a = 10  # a points to the integer object 10
a = 20  # a now points to a new integer object 20

**Behavior:** If you "change" an immutable object, you are actually creating a new object instead of modifying the existing one.

• **Reassignment and Memory Address Change:**• When you reassign an immutable object, such as an integer, string, or tuple, you create a new object. The variable then points to this new object, which is stored at a different memory address.
• **Example:**
a = 10  # a points to memory address for the integer 10
print(id(a))  # Example output: 140634989756128 (some memory address)

a = 20  # Now a points to a new memory address for the integer 20
print(id(a))  # Example output: 140634989756144 (different memory address)

## Mutable Objects

• **Definition:** Mutable objects can be changed after they are created.
**Lists (`list`):** You can change the contents of a list without needing to reassign the list variable. For example:

my_list = [1, 2, 3]  # my_list contains [1, 2, 3]
my_list.append(4)    # my_list is now [1, 2, 3, 4]

**Behavior:** Mutating a mutable object (like appending to a list) modifies the object in place, and all references to that object will reflect the changes.

• **Modification and Same Memory Address:**
  • When you modify a mutable object, such as a list, dictionary, or set, you are changing the object in place. The memory address remains the same because the object itself has not been replaced; its content has just been altered.

• **Example:**
my_list = [1, 2, 3]  # my_list points to some memory address

```
print(id(my_list))  # Example output: 140634989756224

my_list.append(4)    # Modify the list in place
print(my_list)       # Output: [1, 2, 3, 4]
print(id(my_list))  # Memory address remains the same:
140634989756224
```

## Key Takeaway

- For **immutable objects**
  - you must reassign to change their value (like integers or strings).
  - Reassignment creates a new object.
  - The variable points to a different memory address after reassignment.

- For **mutable objects**
  - you can modify their contents directly (like lists or dictionaries) without reassignment.
  - Modifications occur in place, and the object retains the same memory address.
  - No reassignment is necessary to change their contents.

# *Comparison operators*

## 1. Overview of Comparison Operators

**Comparison operators** in Python are used to compare two values. The result of a comparison is always a Boolean value: `True` or `False`.

## 2. List of Comparison Operators

Here's a comprehensive list of comparison operators available in Python:

| Operator | Name | Description |
|---|---|---|
| `==` | Equal to | Checks if the values of two operands are equal |
| `!=` | Not Equal to | Checks if the values of two operands are not equal |
| `>` | Greater Than | Checks if the value on the left is greater than the right |
| `<` | Less Than | Checks if the value on the left is less than the right |
| `>=` | Greater Than or Equal | Checks if the value on the left is greater than or equal to the right |
| `<=` | Less Than or Equal | Checks if the value on the left is less than or equal to the right |
| `is` | Identity Operator | Checks if two operands refer to the same object in memory |
| `is not` | Negative Identity | Checks if two operands do not refer to the same object in memory |
| `in` | Membership Operator | Checks if a value exists within an iterable |
| `not in` | Negative Membership | Checks if a value does not exist within an iterable |

## 3. Detailed Explanation and Examples

# 1. Equal to (==)

**Description:**

Checks if the values of two operands are equal. Returns `True` if they are equal, `False` otherwise.

```
a = 5
b = 5
c = 10

print(a == b)  # Output: True
print(a == c)  # Output: False
```

# 2. Not Equal to (!=)

**Description:**

Checks if the values of two operands are not equal. Returns `True` if they are not equal, `False` otherwise.

```
x = "Python"
y = "Java"
z = "Python"

print(x != y)  # Output: True
print(x != z)  # Output: False
```

# Greater Than (>)

**Description:**
Checks if the value on the left is greater than the value on the right. Returns `True` if it is, `False` otherwise.

```
num1 = 15
num2 = 10

print(num1 > num2)  # Output: True
print(num2 > num1)  # Output: False
```

## 4. Less Than (`<`)

**Description:**
Checks if the value on the left is less than the value on the right. Returns `True` if it is, `False` otherwise.

```
a = 3
b = 7

print(a < b)  # Output: True
print(b < a)  # Output: False
```

## 5. Greater Than or Equal to (`>=`)

**Description:**
Checks if the value on the left is greater than or equal to the

value on the right. Returns `True` if it is, `False` otherwise.

```
x = 20
y = 20
z = 15

print(x >= y)  # Output: True
print(z >= y)  # Output: False
```

6. Less Than or Equal to (`<=`)

**Description:**

Checks if the value on the left is less than or equal to the value on the right. Returns `True` if it is, `False` otherwise.

```
m = 8
n = 12
p = 8

print(m <= n)  # Output: True
print(m <= p)  # Output: True
print(n <= m)  # Output: False
```

## 7. Identity Operators (`is`, `is not`)

**Description:**

- `is` checks if two operands refer to the **same object** in memory.
- `is not` checks if two operands do **not** refer to the same

object in memory.

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a is b)     # Output: True
print(a is c)     # Output: False
print(a is not c)  # Output: True
```

## 8. Membership Operators (`in`, `not in`)

**Description:**

• `in` checks if a value exists within an **iterable** (like a list, tuple, string, etc.). Returns `True` if it does, `False` otherwise.
• `not in` checks if a value does **not** exist within an iterable. Returns `True` if it does not, `False` otherwise.

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)     # Output: True
print("grape" in fruits)      # Output: False
print("grape" not in fruits)  # Output: True
```

Additional Example with Strings:

```
text = "Hello, World!"
print("World" in text)      # Output: True
print("Python" in text)     # Output: False
```

```
print("Python" not in text) # Output: True
```

# 4. Operator Precedence

Understanding **operator precedence** helps predict the order in which operations are evaluated in complex expressions. In Python, comparison operators have a specific precedence level.

**Precedence Hierarchy (Relevant to Comparison Operators):**

1. **Parentheses** `()`
2. **Exponentiation** `**`
3. **Unary plus and minus** `+x`, `-x`
4. **Multiplication**, **Division**, **Floor Division**, **Modulus** `*`, `/`, `//`, `%`
5. **Addition** and **Subtraction** `+`, `-`
6. **Comparison Operators** `==`, `!=`, `>`, `<`, `>=`, `<=`, `is`, `is not`, `in`, `not in`
7. **Logical NOT** `not`
8. **Logical AND** `and`
9. **Logical OR** `or`

Example:

```
result = 3 + 4 > 5  # Evaluated as (3 + 4) > 5
print(result)       # Output: True
```

```
result = 3 + (4 > 5)  # Evaluated as 3 + False => 3 + 0 => 3
print(result)       # Output: 3
```

# 5. Chaining Comparison Operators

Example:

```
x = 5
print(1 < x < 10)  # Output: True
print(5 < x < 10)  # Output: False
```

**Explanation:**

- `1 < x < 10` checks if `x` is greater than `1` **and** less than `10`.
- `5 < x < 10` checks if `x` is greater than `5` **and** less than `10`.

Since `x` is `5`, this evaluates to `False`.

# *Logical operators*

## Overview of Logical Operators

**Logical operators** allow you to combine multiple conditional statements and make complex decisions in your code. Python provides three primary logical operators:

- and

- `or`
- `not`

These operators work with Boolean values (`True` and `False`) and expressions that evaluate to Boolean values.

## List of Logical Operators

| Operator | Name | Description |
|---|---|---|
| `and` | Logical AND | Returns `True` if **both** operands are `True`. |
| `or` | Logical OR | Returns `True` if **at least one** operand is `True`. |
| `not` | Logical NOT | Returns the **opposite** Boolean value of the operand. |

# Detailed Explanation and Examples

### 1. `and` Operator

**Description:**

The `and` operator returns `True` only if **both** operands are `True`. If either operand is `False`, the result is `False`.

**Truth Table:**

| operand1 | operand2 | operand1 and operand2 |
|---|---|---|
| `True` | `True` | `True` |
| `True` | `False` | `False` |
| `False` | `True` | `False` |
| `False` | `False` | `False` |

Example:

```
# Example 1: Both conditions True
age = 25
has_license = True

can_drive = age >= 18 and has_license
print(can_drive)  # Output: True

# Example 2: One condition False
age = 16
has_license = False

can_drive = age >= 18 and has_license
print(can_drive)  # Output: False

# Example 3: Mixed conditions
age = 20
has_license = False

can_drive = age >= 18 and has_license
print(can_drive)  # Output: False
```

## 2. `or` Operator

**Description:**

The `or` operator returns `True` if **at least one** of the operands is `True`. If both operands are `False`, the result is

False.

Truth Table:

| operand1 | operand2 | operand1 or operand2 |
| --- | --- | --- |
| `True` | `True` | `True` |
| `True` | `False` | `True` |
| `False` | `True` | `True` |
| `False` | `False` | `False` |

Example:

```
# Example 1: Both conditions True
is_weekend = True
is_holiday = True

can_sleep_in = is_weekend or is_holiday
print(can_sleep_in)  # Output: True

# Example 2: One condition True
is_weekend = True
is_holiday = False

can_sleep_in = is_weekend or is_holiday
print(can_sleep_in)  # Output: True

# Example 3: Both conditions False
is_weekend = False
is_holiday = False
```

```
can_sleep_in = is_weekend or is_holiday
print(can_sleep_in)  # Output: False
```

## 3. `not` Operator

**Description:**

The `not` operator negates the Boolean value of its operand. If the operand is `True`, `not` returns `False`, and vice versa.

**Truth Table:**

| operand | not operand |
|---|---|
| `True` | `False` |
| `False` | `True` |

```
# Example 1: Negating True
is_sunny = True
is_not_sunny = not is_sunny
print(is_not_sunny)  # Output: False

# Example 2: Negating False
is_raining = False
is_not_raining = not is_raining
print(is_not_raining)  # Output: True

# Example 3: Using with conditions
age = 20
```

has_license = False

can_drive = age >= 18 and has_license
print(can_drive)  # Output: False

can_not_drive = not can_drive
print(can_not_drive)  # Output: True

**4. Operator Precedence**
**Precedence Hierarchy (Relevant to Logical Operators):**
1. **Parentheses** ( )
2. **Unary** not not
3. **Logical** and and
4. **Logical** or or

Example:
# Without parentheses
result = True or False and False
print(result)  # Output: True

# With parentheses to change precedence
result = (True or False) and False
print(result)  # Output: False

# 5. Short-Circuit Evaluation
## a. and Operator:
- **Behavior:**

If the first operand is False, Python **does not evaluate** the second operand because the overall result cannot be True.

**Example:**

```
def first():
    print("First function called")
    return False

def second():
    print("Second function called")
    return True

result = first() and second()
print(result)

# Output:
# First function called
# False
```

## b. or Operator:

- **Behavior:**

If the first operand is True, Python **does not evaluate** the second operand because the overall result is already True.

Example:

```python
def first():
    print("First function called")
    return True

def second():
    print("Second function called")
    return False

result = first() or second()
print(result)

# Output:
# First function called
# True
```

## 6. Truthy and Falsy Values

In Python, values are inherently classified as **truthy** or **falsy** based on their Boolean value. Understanding this classification is essential when using logical operators.

### a. Falsy Values:

These values are considered `False` in Boolean contexts:

- `False`
- `None`
- Zero of any numeric type: `0`, `0.0`, `0j`, etc.
- Empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`,

`range(0)`

- Objects of classes that implement `__bool__()` or `__len__()` returning `False` or `0`

## b. Truthy Values:

Any value that is **not** falsy is considered `True`:

◇ `True`
◇ Non-zero numbers: `1`, `-1`, `3.14`, etc.
◇ Non-empty sequences and collections: `'a'`, `(1,)`, `[1, 2]`, `{'key': 'value'}`, `{1}`, `range(1)`
◇ Objects of classes that implement `__bool__()` or `__len__()` returning `True` or a positive number

# 7. Using Logical Operators in Control Flow

Logical operators are extensively used in control flow statements like `if`, `elif`, `while`, and more to make decisions based on multiple conditions.

### a. Using `and` in `if` Statements

Example:

```
age = 22
has_license = True

if age >= 18 and has_license:
    print("You are eligible to drive.")
else:
```

    print("You are not eligible to drive.")

**b. Using `or` in `if` Statements**

Example:

```
is_weekend = False
is_holiday = True

if is_weekend or is_holiday:
    print("You can relax today!")
else:
    print("Time to work!")
```

**c. Using `not` in `if` Statements**

Example:

```
is_raining = False

if not is_raining:
    print("You don't need an umbrella today.")
else:
    print("Don't forget your umbrella!")
```

**d. Combining Multiple Logical Operators**

```
age = 30
has_license = True
is_insured = False

if (age >= 18 and has_license) or is_insured:
```

```
    print("Eligible for the driving test.")
else:
    print("Not eligible for the driving test.")
```

# 8. Advanced Topics

**a. Combining Logical Operators with Other Operators**

Logical operators can be combined with comparison operators, membership operators, and more to form complex conditions.

Example:

```
username = "admin"
password = "password123"

if username == "admin" and password == "password123":
    print("Access granted.")
else:
    print("Access denied.")
```

## b. Logical Operators in List Comprehensions

Example:

```
numbers = range(1, 21)
filtered = [num for num in numbers if num % 2 == 0 and
num % 3 == 0]
print(filtered)  # Output: [6, 12, 18]
```

## c. Using Logical Operators with Functions

You can use logical operators to combine multiple function calls that return Boolean values.

Example:

```
def is_even(num):
    return num % 2 == 0

def is_positive(num):
    return num > 0

number = 4

if is_even(number) and is_positive(number):
    print("Number is positive and even.")
else:
    print("Number does not meet the criteria.")
```

## d. Ternary Conditional Operator with Logical Operators

Combine logical operators with Python's ternary conditional operator for concise conditional expressions.

Example:

```
age = 20
```

```
has_permission = True

status = "Allowed" if age >= 18 and has_permission else "Not Allowed"
print(status)  # Output: Allowed
```

# ***Comments***

Types of Comments in Python
Python supports several types of comments, each serving different purposes. Understanding these types will help you effectively annotate your code.

## 1. Single-Line Comments

Syntax:

```
# This is a single-line comment
```

Example:

```
# Calculate the area of a circle
radius = 5
area = 3.14159 * radius ** 2  # Area formula: πr²
print(area)
```

### 2. Multi-Line Comments

Approach 1: Multiple Single-Line Comments

```
# This is a multi-line comment.
# It spans several lines.
# Each line starts with a hash symbol.
```

Approach 2: Multi-Line Strings (Not Recommended for Comments)

```
"""
This is a multi-line string,
not a true comment.
It can be used as a comment,
but it's intended for docstrings.
"""
```

**Note:**
While multi-line strings can be used as comments, it's generally recommended to use multiple single-line comments for clarity and to adhere to best practices.

## 3. Docstrings

**Definition:**
**Docstrings** (documentation strings) are multi-line comments used to document modules, classes, functions, and methods. Unlike regular comments, docstrings are accessible at runtime via the __doc__ attribute and are used by documentation tools.

**Syntax:**

• Enclosed in triple quotes (`"""` or `'''`).

• Placed immediately after the definition of a function, class, or module.

```
def greet(name):
    """
    Greet the user by name.

    Parameters:
    name (str): The name of the user.

    Returns:
    str: A greeting message.
    """
    return f"Hello, {name}!"
```

**Explanation:**

• The docstring provides a description of what the function does, its parameters, and its return value.

• Tools like `help()` and documentation generators (e.g., Sphinx) utilize docstrings to create documentatio

**Accessing Docstrings:**

```
print(greet.__doc__)
```

Output:
Greet the user by name.

Parameters:
name (str): The name of the user.

Returns:
str: A greeting message.

# *List*

**Lists** are one of the most versatile and commonly used data structures in Python. They allow you to store, organize, and manipulate collections of items efficiently. Whether you're dealing with numbers, strings, or even other lists, Python lists provide a flexible way to handle data.

## *Declaring Lists*

A **list** in Python is an ordered, mutable (changeable) collection of items. Lists are defined by enclosing elements within square brackets `[]`, separated by commas.

Examples:

```
# An empty list
empty_list = []

# List of integers
numbers = [1, 2, 3, 4, 5]

# List of strings
fruits = ["apple", "banana", "cherry"]

# Mixed data types
mixed = [1, "hello", 3.14, True]

# List of lists (nested list)
nested_list = [[1, 2], [3, 4], [5, 6]]
```

## Creating Lists Using the `list()` Constructor:

```
# From a tuple
tuple_data = (1, 2, 3)
list_from_tuple = list(tuple_data)
```

```
print(list_from_tuple)  # Output: [1, 2, 3]

# From a string
string_data = "hello"
list_from_string = list(string_data)
print(list_from_string)  # Output: ['h', 'e', 'l', 'l', 'o']
```

## *Accessing Items*

You can access items in a list by referring to their **index**. Python uses **zero-based indexing**, meaning the first item has an index of `0`, the second item has an index of `1`, and so on.

### Examples:

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]

# Access the first item
print(fruits[0])  # Output: apple

# Access the third item
```

print(fruits[2])  # Output: cherry

# Access the last item using negative indexing
print(fruits[-1])  # Output: elderberry

# Access the second-to-last item
print(fruits[-2])  # Output: date

Handling Index Errors:
print(fruits[10])  # Raises IndexError: list index out of range
**Tip:** To avoid errors, ensure that the index is within the valid range using `len()` or try-except blocks.

## *Changing Items*

Lists are **mutable**, meaning you can change their content after creation by assigning a new value to a specific index.

### Examples:

numbers = [10, 20, 30, 40, 50]

# Change the first item

```python
numbers[0] = 15
print(numbers)  # Output: [15, 20, 30, 40, 50]

# Change the last item using negative indexing
numbers[-1] = 55
print(numbers)  # Output: [15, 20, 30, 40, 55]

# Change a middle item
numbers[2] = 35
print(numbers)  # Output: [15, 20, 35, 40, 55]
```

# *Removing List Items*

Removing List Items
Python provides several methods to remove items from a list. The choice of method depends on whether you know the item's index, the item's value, or need to remove items based on a condition

### a. `remove()` Method

Syntax:
my_list.remove(value)

Removes the **first occurrence** of a specified value.
fruits = ["apple", "banana", "cherry", "banana", "date"]

**Example:**
# Remove the first 'banana'
fruits.remove("banana")
print(fruits)  # Output: ['apple', 'cherry', 'banana', 'date']

**Note:** If the value is not found, a `ValueError` is raised.

## b. `pop()` Method

Removes an item at a specified index and **returns** it. If no index is specified, it removes and returns the **last item**.

**Syntax:**
item = my_list.pop(index)

Examples:
numbers = [10, 20, 30, 40, 50]

# Remove and return the item at index 2
removed_item = numbers.pop(2)

```
print(removed_item)  # Output: 30
print(numbers)       # Output: [10, 20, 40, 50]

# Remove and return the last item
last_item = numbers.pop()
print(last_item)  # Output: 50
print(numbers)    # Output: [10, 20, 40]
```

### c. `del` Statement

Removes an item or a **slice** from the list. It does not return the removed item.

**Syntax:**

```
del my_list[index]
del my_list[start:end]
```

**Examples:**

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]

# Remove the item at index 1
del fruits[1]
print(fruits)  # Output: ['apple', 'cherry', 'date', 'elderberry']

# Remove a slice (first two items)
del fruits[0:2]
print(fruits)  # Output: ['date', 'elderberry']
```

### d. `clear()` Method

Removes **all items** from the list, resulting in an empty list.

**Syntax:**
my_list.clear()

**Example:**
numbers = [1, 2, 3, 4, 5]
numbers.clear()
print(numbers)  # Output: []

# *Indexing*

**Indexing** allows you to access elements in a list based on their position. Python supports both **positive** and **negative** indexing.

**a. Positive Indexing: Starts from `0` (first element) to `n-1` (last element).**

Example:
```
colors = ["red", "green", "blue", "yellow", "purple"]

print(colors[0])  # Output: red
print(colors[2])  # Output: blue
print(colors[4])  # Output: purple
```

**b. Negative Indexing: Starts from -1 (last element) to -n (first element).**

Example:
```
colors = ["red", "green", "blue", "yellow", "purple"]

print(colors[-1])  # Output: purple
print(colors[-3])  # Output: blue
print(colors[-5])  # Output: red
```

**c. Accessing Elements in Nested Lists**

**Example:**
```
nested_list = [
    [1, 2, 3],
    ["a", "b", "c"],
    [True, False]
]

print(nested_list[0][1])  # Output: 2
print(nested_list[1][2])  # Output: c
```

print(nested_list[2][0])  # Output: True

# *Slicing*

**Slicing** allows you to access a **subset** of a list by specifying a **range** of indices. The syntax uses the colon `:` operator.

Syntax:
subset = my_list[start:stop:step]

- **start:** Starting index (inclusive). Defaults to `0` if omitted.
- **stop:** Ending index (exclusive). Defaults to the end of the list if omitted.
- **step:** Step size. Defaults to `1` if omitted.

Examples:
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Get items from index 2 to 5
subset = numbers[2:6]
print(subset)  # Output: [2, 3, 4, 5]

```
# Get items from the beginning to index 4
subset = numbers[:5]
print(subset)  # Output: [0, 1, 2, 3, 4]

# Get items from index 5 to the end
subset = numbers[5:]
print(subset)  # Output: [5, 6, 7, 8, 9]

# Get every second item from index 1 to 7
subset = numbers[1:8:2]
print(subset)  # Output: [1, 3, 5, 7]

# Reverse the list using slicing
reversed_list = numbers[::-1]
print(reversed_list)  # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

**Slicing with Negative Indices:**

```
colors = ["red", "green", "blue", "yellow", "purple"]

# Get last two items
last_two = colors[-2:]
print(last_two)  # Output: ['yellow', 'purple']

# Get items from index -4 to -1
subset = colors[-4:-1]
print(subset)  # Output: ['green', 'blue', 'yellow']
```

## Note:

• The `stop` index is **exclusive**, meaning the element at that index is **not** included in the result.
• If `start` is greater than `stop` with a positive `step`, the result is an empty list.

### 5. Summary of Default Values in Slicing

| Slicing Syntax | Step | Default Start | Default Stop |
|---|---|---|---|
| `list[start:stop:step]` | Positive | `0` | `len(list)` |
| `list[start:stop:step]` | Negative | `len(list) - 1` | `-1` (exclusive) |

### 5. Summary of Default Values in Slicing

| Slicing Syntax | Step | Default Start | Default Stop |
|---|---|---|---|
| `list[start:stop:step]` | Positive (`> 0`) | `0` | `len(list)` |
| `list[start:stop:step]` | Negative (`< 0`) | `-1` (or `len(list) - 1`) | `-(len(list) + 1)` (exclusive) |

```
# Important Note
# //print(numbers[-1 or 9 is same ::-1])
#
# in stop: -1 (If I put it, Python Thinking the last element =-1)
# So It print: []
#
#  start: -1[inclusive]
```

```
#  end: -1 [execlusive]
#
#  There is nothing; that's why it print []
```

Example:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numbers[-1::-1])
print(numbers[len(numbers)-1::-1])
print(numbers[::1])
```

# *Sorting Lists*

Sorting arranges the elements of a list in a specified order, either **ascending** or **descending**. Python provides built-in methods to sort lists.

## a. `sort()` Method

Sorts the list **in place**, modifying the original list.

Syntax:

```
my_list.sort(reverse=False, key=None)
```

- **reverse (bool):** If `True`, sort the list in **descending** order.

• **key (function):** A function that serves as a **key** for the sort comparison.

**Examples:**

```
# Sorting a list of numbers in ascending order
numbers = [5, 2, 9, 1, 5, 6]
numbers.sort()
print(numbers)  # Output: [1, 2, 5, 5, 6, 9]

# Sorting in descending order
numbers.sort(reverse=True)
print(numbers)  # Output: [9, 6, 5, 5, 2, 1]

# Sorting a list of strings
fruits = ["banana", "apple", "cherry", "date"]
fruits.sort()
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'date']

# Sorting based on the length of strings
fruits.sort(key=len)
print(fruits)  # Output: ['date', 'apple', 'banana', 'cherry']
```

Examples:

```
# Sorting a list of numbers in ascending order
numbers = [5, 2, 9, 1, 5, 6]
numbers.sort()
print(numbers)  # Output: [1, 2, 5, 5, 6, 9]
```

```
# Sorting in descending order
numbers.sort(reverse=True)
print(numbers)  # Output: [9, 6, 5, 5, 2, 1]

# Sorting a list of strings
fruits = ["banana", "apple", "cherry", "date"]
fruits.sort()
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'date']

# Sorting based on the length of strings
fruits.sort(key=len)
print(fruits)  # Output: ['date', 'apple', 'banana', 'cherry']
```

## b. `sorted()` Function

Returns a **new sorted list** without modifying the original list.

Syntax:

```
sorted_list = sorted(my_list, reverse=False, key=None)
```

**Examples:**

```
numbers = [3, 1, 4, 1, 5, 9]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # Output: [1, 1, 3, 4, 5, 9]
print(numbers)         # Output: [3, 1, 4, 1, 5, 9] (original list
```

remains unchanged)

# Sorting strings in descending order
fruits = ["banana", "apple", "cherry", "date"]

sorted_fruits = sorted(fruits, reverse=True)
print(sorted_fruits)  # Output: ['date', 'cherry', 'banana', 'apple']

**Sorting with Keys**

To sort based on multiple criteria, the key function can return a tuple.

Example:
# Sorting a list of tuples based on the second element
students = [("Alice", 25), ("Bob", 20), ("Charlie", 23)]
students.sort(key=lambda student: student[1])
print(students)  # Output: [('Bob', 20), ('Charlie', 23), ('Alice', 25)]

**`lambda student: student[1]`**:

• This is a small anonymous function defined using `lambda`.
• It takes a single argument `student`, which represents each tuple in the `students` list.
• `student[1]` returns the second element of the tuple

(which is the age).

## Sorting Order

• The `sort()` method arranges the tuples in ascending order based on the values returned by the key function. So the sorting will work as follows:

◇ Compare ages: `20`, `23`, `25`

◇ Sorted order will be:

■ `("Bob", 20)`

■ `("Charlie", 23)`

■ `("Alice", 25)`

## d. Sorting with Multiple Keys

To sort based on multiple criteria, the key function can return a tuple.

Example:

# Sorting by age, then by name
people = [

```
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 20},
    {"name": "Charlie", "age": 25},
    {"name": "Dave", "age": 20}
]
```

```
# Sort first by age, then by name
people_sorted = sorted(people, key=lambda person:
(person["age"], person["name"]))
print(people_sorted)
# Output:
# [
#     {'name': 'Bob', 'age': 20},
#     {'name': 'Dave', 'age': 20},
#     {'name': 'Alice', 'age': 25},
#     {'name': 'Charlie', 'age': 25}
# ]
```

## *Joining Lists*

Joining (or concatenating) lists combines two or more lists into a single list. Python provides several ways to achieve this.

## a. Using the + Operator

The + operator concatenates two lists, returning a new list.

**Syntax:**
combined_list = list1 + list2

Example:
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined = list1 + list2
print(combined)  # Output: [1, 2, 3, 4, 5, 6]

**b. Using the extend() Method**

The extend() method appends all elements from another list to the **end** of the current list. This modifies the original list in place.

Syntax:
list1.extend(list2)

Example:

```
list1 = ["a", "b", "c"]
list2 = ["d", "e", "f"]
list1.extend(list2)
print(list1)  # Output: ['a', 'b', 'c', 'd', 'e', 'f']
```

**c. Using the * Operator (Replicating Lists)**

The * operator can be used to **repeat** a list multiple times.

Syntax:

```
repeated_list = my_list * n
```

Example:

```
letters = ["x", "y"]
repeated = letters * 3
print(repeated)  # Output: ['x', 'y', 'x', 'y', 'x', 'y']
```

**d. Using List Comprehensions**

List comprehensions can be used to combine lists in more complex ways.

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

# Combine lists with list comprehension
combined = [item for sublist in [list1, list2] for item in sublist]
print(combined)  # Output: [1, 2, 3, 4, 5, 6]
```

**e. Using the `itertools` Module**
For more advanced list joining, the `itertools` module provides tools like `chain()`.

Example:
```
import itertools

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]

combined = list(itertools.chain(list1, list2, list3))
print(combined)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# *Appending to Lists*

Appending adds a new element to the **end** of a list. Python provides the `append()` and `insert()` methods for adding elements.

## a. `append()` Method
Adds a single element to the end of the list.
Syntax:
```
my_list.append(element)
```

Example:
fruits = ["apple", "banana", "cherry"]
fruits.append("date")
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'date']

## b. `insert()` Method

Inserts an element at a specified index.

Syntax:
my_list.insert(index, element)

Example:
fruits = ["apple", "banana", "cherry"]
fruits.insert(1, "blueberry")  # Insert at index 1
print(fruits)  # Output: ['apple', 'blueberry', 'banana', 'cherry']

## `extend()` Method

As discussed earlier, `extend()` adds multiple elements from another list.

Example:
numbers = [1, 2, 3]
numbers.extend([4, 5, 6])

print(numbers)  # Output: [1, 2, 3, 4, 5, 6]

## *Counting Item*

Counting the number of occurrences of a specific element in a list can be done using the `count()` method.

**Syntax:**

count = my_list.count(element)

Example:

numbers = [1, 2, 3, 2, 4, 2, 5]

# Count occurrences of 2

count_twos = numbers.count(2)

print(count_twos)  # Output: 3

# Count occurrences of 6 (which is not in the list)

count_sixes = numbers.count(6)

print(count_sixes)  # Output: 0

**Use Case:**
Counting items is useful for frequency analysis, statistical calculations, and conditional operations based on the number of occurrences.

# *Additional List Operations*

## a. `index()` Method

Finds the **first index** of a specified value. Raises a `ValueError` if the value is not found.

**Syntax:**
index = my_list.index(value, start, end)

- **start (optional):** Start searching from this index.
- **end (optional):** Stop searching before this index.

```
fruits = ["apple", "banana", "cherry", "banana", "date"]

# Find the index of 'banana'
index_banana = fruits.index("banana")
print(index_banana)  # Output: 1
```

# Find the index of 'banana' starting from index 2
index_banana_from_2 = fruits.index("banana", 2)
print(index_banana_from_2)  # Output: 3

## b. reverse() Method

Reverses the elements of the list **in place**.

Syntax:
my_list.reverse()

Example:
numbers = [1, 2, 3, 4, 5]
numbers.reverse()
print(numbers)  # Output: [5, 4, 3, 2, 1]

### c. copy() Method

Creates a **shallow copy** of the list.

Syntax
new_list = my_list.copy()

**Example:**
original = [1, 2, 3]
copied = original.copy()
copied.append(4)

print(original)  # Output: [1, 2, 3]
print(copied)    # Output: [1, 2, 3, 4]

### d. `clear()` Method

Removes **all items** from the list, resulting in an empty list.

Syntax:
my_list.clear()

Example:
fruits = ["apple", "banana", "cherry"]
fruits.clear()
print(fruits)  # Output: []

### e. `extend()` vs `append()`

- `append()`**:** Adds its argument as a single element to the end of the list.
- `extend()`**:** Iterates over its argument and adds each element to the list, extending it.

Example:
list1 = [1, 2, 3]

# Using append()
list1.append([4, 5])

```
print(list1)  # Output: [1, 2, 3, [4, 5]]

# Reset list1
list1 = [1, 2, 3]

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

# *List Comprehensions*

**List comprehensions** provide a concise way to create lists based on existing lists. They can include conditional logic and transformations.

**Basic Syntax:**

```
new_list = [expression for item in iterable if condition]
```

**Examples:**

```
# Create a list of squares
squares = [x**2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Create a list of even numbers
```

```
evens = [x for x in range(20) if x % 2 == 0]
print(evens)  # Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# Create a list of uppercase fruits
fruits = ["apple", "banana", "cherry"]
uppercase_fruits = [fruit.upper() for fruit in fruits]
print(uppercase_fruits)  # Output: ['APPLE', 'BANANA',
'CHERRY']

# Nested list comprehension
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]
print(flattened)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# *Nested Lists*

**Nested lists** are lists within lists, allowing you to create multi-dimensional data structures.

**Examples:**

```
# Creating a nested list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
```

```
    [7, 8, 9]
]

print(matrix)
# Output:
# [
#     [1, 2, 3],
#     [4, 5, 6],
#     [7, 8, 9]
# ]

# Accessing elements in a nested list
print(matrix[0][1])  # Output: 2 (first row, second column)
print(matrix[2][0])  # Output: 7 (third row, first column)

# Modifying an element
matrix[1][2] = 60
print(matrix)
# Output:
# [
#     [1, 2, 3],
#     [4, 5, 60],
#     [7, 8, 9]
# ]
```

# *Copying Lists*

## Copying Lists

Copying lists involves creating a new list that contains the same elements as the original. Python provides different ways to copy lists, each with its own implications

**a. Shallow Copy**

A **shallow copy** creates a new list object but **does not** create copies of nested objects. Instead, it references the same nested objects.

**Methods to Create a Shallow Copy:**

1. Using the `copy()` method
2. Using slicing `[:]`
3. Using the `list()` constructor

**Examples:**

import copy

original = [1, 2, [3, 4]]

# Using the copy() method
shallow_copy1 = original.copy()

# Using slicing
shallow_copy2 = original[:]

```
# Using the list() constructor
shallow_copy3 = list(original)

print(shallow_copy1)  # Output: [1, 2, [3, 4]]
print(shallow_copy2)  # Output: [1, 2, [3, 4]]
print(shallow_copy3)  # Output: [1, 2, [3, 4]]

# Modifying a nested list in the shallow copy affects the
original
shallow_copy1[2].append(5)
print(original)       # Output: [1, 2, [3, 4, 5]]
print(shallow_copy1)  # Output: [1, 2, [3, 4, 5]]
```

## b. Deep Copy

A **deep copy** creates a new list and **recursively copies** all nested objects, resulting in complete independence from the original list.

**Method to Create a Deep Copy:**

• Using the `deepcopy()` function from the `copy` module

**Example:**

```
import copy

original = [1, 2, [3, 4]]
```

```
deep_copy = copy.deepcopy(original)

# Modifying the deep copy does not affect the original
deep_copy[2].append(5)
print(original)   # Output: [1, 2, [3, 4]]
print(deep_copy)  # Output: [1, 2, [3, 4, 5]]
```

Differences Between Shallow and Deep Copies

| Feature | Shallow Copy | Deep Copy |
|---|---|---|
| Copies nested objects? | No (references same nested objects) | Yes (creates independent copies) |
| Affects original when nested objects change? | Yes | No |

## Use Cases:

- **Shallow Copy:** When you need a copy of the list structure but are okay with shared nested objects.
- **Deep Copy:** When you require complete independence, especially with complex, nested data structures.

# *Control Flow*

# *Conditional Statements*

## 1. Introduction to Conditional Statements

**Conditional statements** allow your program to execute certain pieces of code based on whether a condition is `True` or `False`. They are essential for creating dynamic programs that can handle different scenarios.

### Why Use Conditional Statements?

- **Decision Making**: Choose different paths of execution based on input or other factors.
- **Control Flow**: Direct the flow of your program to perform tasks only when specific conditions are met.
- **Flexibility**: Make your code adaptable to varying situations.

# *The if Statement*

## The `if` Statement

The `if` statement is the most basic form of a conditional statement. It allows you to execute a block of code only if a specified condition is `True`.

**Syntax**

```
if condition:
    # Code block to execute if condition is True
```

**Example**

```
age = 18

if age >= 18:
    print("You are an adult.")
```

**Output:**

```
You are an adult.
```

# *The if-else Statement*

An `if-else` statement allows you to execute one block of code if a condition is `True` and another block if the condition is `False`.

**Syntax**

```
if condition:
    # Code block if condition is True
else:
    # Code block if condition is False
```

**Example**

```
age = 16

if age >= 18:
    print("You are an adult.")
else:
    print("You are not an adult.")
```

**Output:**

```
You are not an adult.
```

## *The if-elif-else Statement*

The `if-elif-else` statement allows you to check multiple conditions sequentially. Once a condition is `True`, the corresponding block is executed, and the rest are skipped.

**Syntax**

```
if condition1:
   # Code block if condition1 is True
elif condition2:
   # Code block if condition2 is True
elif condition3:
   # Code block if condition3 is True
...
else:
   # Code block if none of the above conditions are True
```

**Example**

```
score = 85

if score >= 90:
   print("Grade: A")
elif score >= 80:
   print("Grade: B")
elif score >= 70:
   print("Grade: C")
else:
   print("Grade: F")
```

# *Nested if Statements*

**Nested `if` statements** are `if` statements placed inside another `if` statement. They allow for more granular decision-making based on multiple layers of conditions.

### Syntax

```
if condition1:
    # Code block if condition1 is True
    if condition2:
        # Code block if condition2 is True
    else:
        # Code block if condition2 is False
else:
    # Code block if condition1 is False
```

**Example**

```
age = 20
has_license = True

if age >= 18:
    print("You are eligible to drive.")
    if has_license:
```

```
        print("You can drive a car.")
    else:
        print("You need a driver's license to drive.")
else:
    print("You are not eligible to drive.")
```

**Output:**
You are eligible to drive.
You can drive a car.

# ***Using Logical Operators in Conditions***

Logical operators allow you to combine multiple conditions within a single `if`, `elif`, or `else` statement.

## Logical Operators

- `and`: Both conditions must be `True`.
- `or`: At least one condition must be `True`.
- `not`: Inverts the truth value of the condition.

## Examples

### a. Using and

```
age = 25
has_license = True

if age >= 18 and has_license:
    print("You can drive a car.")
```

**Output:**

You can drive a car.

### b. Using or

```
is_weekend = True
has_free_time = False

if is_weekend or has_free_time:
    print("You can go out.")
```

**Output:**

You can go out.

### c. Using not

```
is_raining = False

if not is_raining:
    print("You don't need an umbrella.")
```

**Output:**
You don't need an umbrella.

# *Comparison Operators*

Comparison operators are used to compare two values. They return `True` or `False` based on the comparison.

## List of Comparison Operators

| Operator | Description | Example |
|---|---|---|
| `==` | Equal to | `a == b` |
| `!=` | Not equal to | `a != b` |
| `>` | Greater than | `a > b` |
| `<` | Less than | `a < b` |
| `>=` | Greater than or equal to | `a >= b` |
| `<=` | Less than or equal to | `a <= b` |

**Examples**

```
x = 10
y = 20

# Equal to
print(x == y)  # Output: False
```

```
# Not equal to
print(x != y)  # Output: True

# Greater than
print(x > y)   # Output: False

# Less than
print(x < y)   # Output: True

# Greater than or equal to
print(x >= 10) # Output: True

# Less than or equal to
print(y <= 20) # Output: True
```

## *Conditional Expressions (Ternary Operator)*

Python offers a concise way to write conditional statements using **conditional expressions**, also known as the **ternary operator**. This allows you to assign a value to a variable based on a condition in a single line.

**Syntax**
variable = value_if_true if condition else value_if_false

**Example**
```
age = 18
status = "Adult" if age >= 18 else "Minor"
print(status)  # Output: Adult
```

# *Best Practices*

**Avoid Deep Nesting**:
• Excessive nesting can make code hard to read. Consider refactoring using functions or logical operators

```
# Deeply nested
if condition1:
    if condition2:
        if condition3:
            do_something()

# Refactored
if condition1 and condition2 and condition3:
    do_something()
```

**Use `elif` Instead of Multiple `if` Statements**:

• Using `elif` ensures that only one block is executed, improving efficiency.

```
# Using multiple ifs
if x > 0:
    print("Positive")
if x == 0:
    print("Zero")
if x < 0:
    print("Negative")

# Using elif
if x > 0:
    print("Positive")
elif x == 0:
    print("Zero")
else:
    print("Negative")
```

## Common Mistakes

**Using Assignment Instead of Comparison `==`**:

```
age = 18

# Incorrect
```

```python
if age = 18:
    print("You are 18 years old.")

# Correct
if age == 18:
    print("You are 18 years old.")
```

**Logical Errors in Conditions**:

```python
# Intended: Age between 18 and 25
if age >= 18 and age <= 25:
    print("Young adult")

# Common mistake: Using or
if age >= 18 or age <= 25:
    print("Young adult")  # This condition is always True for
age >= 18
```

# *Advanced Topics*

## a. **Chained Comparisons**

Python allows you to chain multiple comparisons in a single statement, making the code more concise and readable.

**Example:**

```
age = 25

if 18 <= age < 30:
    print("Young adult")
```

**Output**:

```
Young adult
```

### b. **Using Functions in Conditions**

Encapsulating conditions within functions can make code more modular and reusable.

**Example:**

```
def is_adult(age):
    return age >= 18

age = 20

if is_adult(age):
    print("You are an adult.")
```

Output:

```
You are an adult.
```

### c. **Short-Circuit Evaluation**

Logical operators in Python use short-circuit evaluation, meaning the second condition is evaluated only if necessary.

**Example:**

```
def check_condition():
    print("Checking condition...")
    return True

if False and check_condition():
    print("This won't print.")
```

**Output:**

# ***Exercise***

## Examples and Exercises

## Example 1: Determine Pass or Fail

**Example:**

```
score = 75

if score >= 60:
    print("Pass")
else:
    print("Fail")
```

**Output:**

```
Pass
```

## Example 2: Grading System

```
score = 88

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
```

```
    grade = "D"
else:
    grade = "F"

print(f"Grade: {grade}")
```

**Output:**

```
Grade: B
```

## Example 3: Nested `if` Statements

```
age = 22
has_license = True

if age >= 18:
    print("Eligible to drive.")
    if has_license:
        print("You can drive a car.")
    else:
        print("You need a driver's license to
drive.")
else:
    print("Not eligible to drive.")
```

**Output:**

```
Eligible to drive.
You can drive a car.
```

## Exercise 4: Even or Odd

**Task:** Write a program that checks if a number is even or odd.

```
number = 7

if number % 2 == 0:
    print("Even")
else:
    print("Odd")
```

**Expected Output:**

```
Odd
```

## Exercise 5: Leap Year Checker

**Task:** Determine if a given year is a leap year.

**Rules:**

• A year is a leap year if it is divisible by 4 **and not** divisible by 100, **unless** it is also divisible by 400.

```
year = 2000

if (year % 4 == 0 and year % 100 != 0) or
(year % 400 == 0):
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

## Exercise 6: Maximum of Three Numbers

**Task:** Find the maximum of three numbers using `if-elif-else`.

```
a = 10
b = 25
c = 20

if a >= b and a >= c:
    max_num = a
elif b >= a and b >= c:
    max_num = b
else:
    max_num = c
```

```python
print(f"The maximum number is {max_num}.")
```

**Expected Output:**

```
The maximum number is 25.
```

# *Loop Statements*

Loops are a fundamental concept in Python (and programming in general) that allow you to execute a block of code multiple times. They are essential for tasks that require repetition, such as iterating over items in a list, processing data, or performing repetitive calculations.

In Python, there are two primary types of loops:

1. `for` **Loops**
2. `while` **Loops**

Additionally, Python provides **loop control statements** that modify the behavior of loops:

- `break`
- `continue`

- `pass`

## *The for Loop*

The `for` loop in Python is used for iterating over a sequence (such as a list, tuple, dictionary, set, or string). It allows you to execute a block of code once for each item in the sequence.

**Syntax**

```
for variable in sequence:
    # Code block to execute for each item
```

- `variable`: A name you choose to refer to the current item in the sequence.
- `sequence`: The collection of items you want to iterate over.

**Example**

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

for fruit in fruits:
    print(fruit)
```

**Output:**
apple
banana
cherry

## Iterating Over Different Data Types

### a. **List**

```
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    print(num)
```

**Output:**
1
2
3
4
5

### b. **Tuple**

```
colors = ("red", "green", "blue")

for color in colors:
    print(color)
```

**Output:**
red
green
blue

c. **Dictionary**
When iterating over a dictionary, the default behavior is to iterate over its keys.

```
student_ages = {"Alice": 25, "Bob": 22, "Charlie": 23}

for student in student_ages:
    print(student)
```

**Output:**
Alice
Bob
Charlie

To iterate over both keys and values:

```
for student, age in student_ages.items():
    print(f"{student} is {age} years old.")
```

**Output:**
Alice is 25 years old.

Bob is 22 years old.
Charlie is 23 years old.

### d. **String**

Iterating over a string allows you to access each character individually.

```
word = "Python"

for letter in word:
    print(letter)
```

**Output**

```
P
y
t
h
o
n
```

**Using `range()` with `for` Loops**

The `range()` function generates a sequence of numbers, which is often used with `for` loops for iteration a specific number of times.

a. **Basic Usage**

```
for i in range(5):
    print(i)
```

**Output:**

```
0
1
2
3
4
```

b. **Specifying Start and End**

```
for i in range(2, 6):
    print(i)
```

**Output:**

```
for i in range(2, 6):
    print(i)
```

c. **Specifying Step**

```
for i in range(0, 10, 2):
    print(i)
```

**Output:**

0
2
4
6
8

**Nested for Loops**

You can place one for loop inside another to handle multi-dimensional data structures.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in matrix:
    for num in row:
        print(num, end=' ')
    print()  # Newline after each row
```

Output:

1 2 3
4 5 6
7 8 9

# *The while Loop*

The `while` loop in Python repeatedly executes a block of code as long as a given condition is `True`.

**Syntax**

```
while condition:
    # Code block to execute repeatedly
```

`condition`: A boolean expression that is evaluated before each iteration. If `True`, the loop continues; if `False`, the loop stops

**Example**

```
count = 0

while count < 5:
    print(count)
    count += 1  # Increment count to eventually break the loop
```

**Output:**

```
0
1
```

2
3
4

Preventing Infinite Loops
An **infinite loop** occurs when the loop's condition never becomes `False`. To prevent this, ensure that the condition will eventually be met.

## Example of an Infinite Loop (Avoid!)

```
count = 0

while count < 5:
    print(count)
    count += 1
```

**Nested `while` Loops**
Similar to nested `for` loops, you can nest `while` loops to handle more complex scenarios.

```
i = 1

while i <= 3:
    j = 1
    while j <= 2:
        print(f"i={i}, j={j}")
        j += 1
```

    i += 1

**Output:**
i=1, j=1
i=1, j=2
i=2, j=1
i=2, j=2
i=3, j=1
i=3, j=2

# *Loop Control Statements*

Loop control statements alter the behavior of loops. Python provides three primary loop control statements:

- `break`
- `continue`
- `pass`

## `break`

The `break` statement terminates the nearest enclosing loop, causing the program to exit the loop immediately.

Example
for num in range(10):

```
    if num == 5:
        break  # Exit the loop when num is 5
    print(num)
```

**Output:**

0

1

2

3

4

## `continue`

The `continue` statement skips the rest of the code inside the loop for the current iteration and moves to the next iteratio

**Example**

```
for num in range(5):
    if num == 2:
        continue  # Skip the rest when num is 2
    print(num)
```

**Output:**

0

1

3

4

`pass`
The `pass` statement is a **null operation**; it does nothing. It is useful as a placeholder when a statement is syntactically required but no action is needed.

# *Loop else Clause*

## Loop `else` Clause
In Python, loops (`for` and `while`) can have an `else` clause. The code inside the `else` block is executed **only if the loop completes normally** (i.e., not terminated by a `break` statement).

### Using `else` with `for` Loops

**Example Without `break`**
```
for num in range(3):
    print(num)
else:
    print("Loop completed without break.")
```

**Output:**
0
1
2
Loop completed without break.

## Example With `break`

```
for num in range(5):
    print(num)
    if num == 2:
        break
else:
    print("Loop completed without break.")
```

**Output:**
0
1
2

**Explanation:**
• Since the loop is terminated by `break`, the `else` block is **n-ot** executed.

## Using `else` with `while` Loops

Example Without break

```
count = 0

while count < 3:
    print(count)
    count += 1
else:
    print("While loop completed without break.")
```

**Output:**

```
0
1
2
While loop completed without break.
```

**Example With break**

```
count = 0

while count < 5:
    print(count)
    if count == 3:
        break
    count += 1
else:
```

print("While loop completed without break.")

**Output:**

0

1

2

3

**Explanation:**

• The loop is terminated by `break`, so the `else` block is **not** executed.

## *Best Practices*

• **Choose the Right Loop Type**:

  • Use `for` loops when iterating over a sequence or when the number of iterations is known.

  • Use `while` loops when the number of iterations is not known in advance and depends on a condition.

• **Avoid Infinite Loops**:

  ◇ Ensure that the loops condition will eventually become `False`.

◇ If using `while True`, make sure there is a `break` statement within the loop.

**Use `enumerate()` and `zip()` When Appropriate**:

- `enumerate()` provides both the index and the value when iterating.
- `zip()` allows parallel iteration over multiple sequences.

Example Using `enumerate()`:

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

for index, fruit in enumerate(fruits):
    print(f"Fruit {index}: {fruit}")
```

**Output:**

Fruit 0: apple
Fruit 1: banana
Fruit 2: cherry

**Example Using `zip()`:**

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 22, 23]
```

```python
for name, age in zip(names, ages):
    print(f"{name} is {age} years old.")
```

**Output:**
Alice is 25 years old.
Bob is 22 years old.
Charlie is 23 years old.

# *Examples and Exercises*

## Example 1: Printing Numbers from 1 to 5

### Using `for` Loop

```python
for num in range(1, 6):
    print(num)
```

### Using `while` Loop

```python
count = 1

while count <= 5:
```

```
    print(count)
    count += 1
```

**Output:**

1
2
3
4
5

## Example 2: Summing Numbers in a List

### Using `for` Loop

```
numbers = [1, 2, 3, 4, 5]
total = 0

for num in numbers:
    total += num

print(f"Total: {total}")
```

**Using `while` Loop**

```
numbers = [1, 2, 3, 4, 5]
total = 0
```

```
index = 0

while index < len(numbers):
    total += numbers[index]
    index += 1

print(f"Total: {total}")
```

**Output:**

Total: 15

### Example 3: **Finding a Specific Element**

**Using `for` Loop with `break`**

```
fruits = ["apple", "banana", "cherry", "date"]

for fruit in fruits:
    if fruit == "cherry":
        print("Cherry found!")
        break
else:
    print("Cherry not found.")
```

**Output:**

Cherry found!

## Exercise 4: Multiplication Table

**Task:** Write a program that prints the multiplication table for numbers 1 through 5.

**Solution Using for Loop**

```
for i in range(1, 6):
    for j in range(1, 11):
        product = i * j
        print(f"{i} x {j} = {product}")
    print("-" * 15)  # Separator after each table
```

**Output:**

```
1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
---------------
2 x 1 = 2
...
5 x 10 = 50
---------------
```

**Exercise 5: Prime Number Checker**

```
num = 29
is_prime = True
```

```
if num <= 1:
    is_prime = False
else:
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            is_prime = False
            break

if is_prime:
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

**Output:**
29 is a prime number.

## Exercise 6: FizzBuzz

**Task:** For numbers from 1 to 15, print "Fizz" for multiples of 3, "Buzz" for multiples of 5, and "FizzBuzz" for multiples of both.

### Solution Using `for` Loop and `if-elif-else`

```
for num in range(1, 16):
    if num % 3 == 0 and num % 5 == 0:
```

```
        print("FizzBuzz")
    elif num % 3 == 0:
        print("Fizz")
    elif num % 5 == 0:
        print("Buzz")
    else:
        print(num)
```

**Output:**

1

2

Fizz

4

Buzz

Fizz

7

8

Fizz

Buzz

11

Fizz

13

14

FizzBuzz

**Exercise 7: Factorial Calculator**

**Task:** Calculate the factorial of a given number.

### Solution Using `while` Loop

```
num = 5
factorial = 1

if num < 0:
    print("Factorial does not exist for negative numbers.")
elif num == 0:
    print("Factorial of 0 is 1.")
else:
    while num > 0:
        factorial *= num
        num -= 1
    print(f"Factorial is {factorial}")
```

**Output:**

Factorial is 120

# *List Comprehensions and*

# *Generator Expressions*

In Python, **List Comprehensions** and **Generator Expressions** provide a concise way to create and manipulate lists and generators. They are not only syntactically elegant but also often more efficient than traditional loops.
• **List Comprehensions**: Generate new lists by applying an expression to each item in an existing iterable, optionally filtering items based on a condition.
• **Generator Expressions**: Similar to list comprehensions but generate items one at a time and are more memory-efficient, especially for large datasets.

# *List Comprehensions*

List comprehensions offer a concise way to create lists. They consist of brackets containing an expression followed by a `for` clause, and optionally, one or more `if` clauses.

Basic Syntax
[expression for item in iterable if condition]

• **expression**: The value or transformation to apply to each item.
• **item**: The variable representing each element in the iterable.
• **iterable**: The collection of items to iterate over (e.g., list, tuple, range).
• **condition** (optional): A filter that determines whether the `expression` is applied to the `item`.

**Examples**

## a. Creating a List of Squares

```
squares = [x**2 for x in range(10)]
print(squares)
```

**Output:**
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

**b. Filtering Even Numbers**

```
even_numbers = [x for x in range(20) if x % 2 == 0]
print(even_numbers)
```

**Output:**
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

**c. Applying a Function to Each Item**

```
def square(x):
    return x * x

squared = [square(x) for x in range(5)]
print(squared)
```

**Output:**

[0, 1, 4, 9, 16]

**d. Flattening a Nested List**

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

flattened = [num for row in matrix for num in row]
print(flattened)
```


```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

flattened = [num for row in matrix for num in row]
print(flattened)
```

**Output:**
[1, 2, 3, 4, 5, 6, 7, 8, 9]

**Nested List Comprehensions:**
List comprehensions can be nested to handle multi-dimensional data structures.

## Example: Transposing a Matrix

List comprehensions can be nested to handle multi-dimensional data structures.


## Example: Transposing a Matrix


**Order of `for` Clauses:**
• In **single list comprehensions** with multiple `for` clauses, the `for` clauses are ordered from **outer to inner**, just like nested `for` loops.

**Syntax**

```
[expr for x in outer for y in inner]  # x is outer, y is inner
```

• In **nested list comprehensions**, each list comprehension can have its own `for` clauses, with the outer list comprehension controlling the outer loop and the inner list comprehension controlling the inner loop.

**Syntax**
[[expr for inner_var in inner_iterable] for outer_var in outer_iterable]

# *Generator Expressions*

Generator expressions are similar to list comprehensions but use parentheses `()` instead of brackets `[]`. They generate items one at a time and are more memory-efficient, making them suitable for large datasets.

**Basic Syntax**
(expression for item in iterable if condition)

## Examples

### a. Creating a Generator for Squares

```
squares_gen = (x**2 for x in range(10))
print(squares_gen)
```

**Output**

```
<generator object <genexpr> at 0x...>
```

To retrieve the items, you can iterate over the generator:

```
for square in squares_gen:
    print(square)
```

Output:

```
0
1
4
9
16
25
36
49
64
81
```

**b. Filtering with a Generator Expression**

```
even_gen = (x for x in range(20) if x % 2 == 0)
for even in even_gen:
    print(even)
```

**Output:**
0
2
4
6
8
10
12
14
16
18

### c. Using `next()` with Generators

You can manually retrieve items using the `next()` function.
squares_gen = (x**2 for x in range(5))
print(next(squares_gen))  # Output: 0
print(next(squares_gen))  # Output: 1

Output:
0
1

**Note:** Once a generator is exhausted, subsequent calls to `next()` will raise a `StopIteration` exception.

## d. Memory Efficiency Example

Creating a large list vs. a generator:

```
# List comprehension
large_list = [x for x in range(1000000)]

# Generator expression
large_gen = (x for x in range(1000000))
```

• **List Comprehension**: Allocates memory for the entire list.
• **Generator Expression**: Generates items on-the-fly, using much less memory.

# ***Generator Memory Efficiency***

Example: Reading a Large File
Suppose you want to read a large file and process each line

**Using List Comprehension**:

```
with open('large_file.txt') as file:
    lines = [line.strip() for line in file]
# All lines are stored in memory
```

**Using Generator Expression**:

```
with open('large_file.txt') as file:
    lines_gen = (line.strip() for line in file)
    for line in lines_gen:
        process(line)  # Process each line one at a time
# Only one line is in memory at a time
```

# *Exercise*

**Example 1: List Comprehension with Condition**

**Task:** Create a list of squares for even numbers between 1 and 10.

```
squares_even = [x**2 for x in range(1, 11) if x % 2 == 0]
print(squares_even)
```

Output:
[4, 16, 36, 64, 100]

**Example 2: Generator Expression for Large Dataset**

**Task:** Create a generator that yields squares of numbers from 1 to 1,000,000.

```
squares_gen = (x**2 for x in range(1, 1000001))
# To demonstrate, we'll print the first 5 squares
for _ in range(5):
    print(next(squares_gen))
```

**Output:**

```
1
4
9
16
25
```

**Note:** Using a generator here avoids storing a large list in memory.

**Example 3: Nested List Comprehension**

**Task:** Create a 3x3 matrix initialized with zeros using a nested list comprehension.

```
rows, cols = 3, 3
matrix = [[0 for _ in range(cols)] for _ in range(rows)]
print(matrix)
```

**Output:**

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

**Exercise 4: Filter and Transform**

**Task:** Given a list of numbers, create a new list containing the cubes of numbers that are divisible by 3.
**List:** `[1, 3, 4, 6, 7, 9, 12]`

**Expected Output:** `[27, 216, 729, 1728]`

Solution Using List Comprehension

```
numbers = [1, 3, 4, 6, 7, 9, 12]
cubes_div3 = [x**3 for x in numbers if x % 3 == 0]
print(cubes_div3)
```

**Exercise 5: Create a Dictionary from Two Lists**
**Task:** Given two lists, one containing names and the other containing ages, create a dictionary that maps each name to its corresponding age.

**Names:** `["Alice", "Bob", "Charlie"]`
**Ages:** `[25, 22, 23]`

**Expected Output:** `{"Alice": 25, "Bob": 22, "Charlie": 23}`

**Solution Using Dictionary Comprehension**

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 22, 23]
age_dict = {name: age for name, age in zip(names, ages)}
print(age_dict)
```

**Output:**
{'Alice': 25, 'Bob': 22, 'Charlie': 23}

**Exercise 6: Prime Number Generator**

## Exercise 7: Prime Number Generator

**Task:** Create a generator expression that yields prime numbers up to 50.

## Solution

Creating a generator expression for prime numbers is more complex because it involves checking each number for primality. Instead, we'll use a generator function with `yield`

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

primes_gen = (x for x in range(2, 51) if is_prime(x))

for prime in primes_gen:
```

```
    print(prime, end=' ')
```

Output:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

**Note:** While list comprehensions can handle this, using a generator expression is more memory-efficient.

## Exercise 8: List Comprehension vs Loop

**Task:** Compare the performance of list comprehensions and `for` loops by creating a list of squares from 1 to 1,000,000.

### Solution

We'll use the `time` module to measure execution time.

```
import time

# Using List Comprehension
start_time = time.time()
squares_list = [x**2 for x in range(1, 1000001)]
end_time = time.time()
print(f"List Comprehension took {end_time - start_time:.4f} seconds.")

# Using For Loop
start_time = time.time()
```

```
squares_loop = []
for x in range(1, 1000001):
    squares_loop.append(x**2)
end_time = time.time()
print(f"For Loop took {end_time - start_time:.4f} seconds.")
```

Sample Output:
List Comprehension took 0.0543 seconds.
For Loop took 0.1521 seconds.

# *Strings*

Strings are one of the most fundamental data types in Python, used extensively for handling and manipulating textual data. Understanding strings and their associated methods is crucial for effective Python programming.

**Table of Contents**

## Introduction to Strings

In Python, a **string** is a sequence of characters enclosed within quotes. Strings are used to represent and manipulate text data.

• **Single Quotes:** `'Hello, World!'`
• **Double Quotes:** `"Hello, World!"`
• **Triple Quotes:** `'''Hello, World!'''` or `"""Hello, World!"""` (used for multi-line strings)

**Key Characteristics:**
◇ **Immutable:** Once created, the contents of a string cannot be changed.
◇ **Ordered:** Characters in a string have a defined order and can be accessed via indices.
◇ **Iterable:** You can iterate over each character in a string using loops.

# *String Basics*

## String Creation

Strings can be created in various ways:
# Using single quotes
string1 = 'Hello, World!'

```
# Using double quotes
string2 = "Hello, World!"

# Using triple quotes for multi-line strings
string3 = """Hello,
World!"""

# Using the str() constructor
string4 = str(123)  # Converts integer to string '123'

print(string1)
print(string2)
print(string3)
print(string4)
```

**Output:**

```
Hello, World!
Hello, World!
Hello,
World!
123
```

## String Immutability

Strings in Python are **immutable**, meaning that once a string is created, its content cannot be changed. Any operation that modifies a string actually creates a new string.

```
s = "Hello"
# Attempting to change the first character
# s[0] = 'h'  # This will raise a TypeError

# Correct way: Create a new string
s_new = 'h' + s[1:]
print(s_new)  # Output: "hello"
```

## String Indexing and Slicing

**Slicing Syntax:**
string[start:stop:step]
• **start**: Starting index (inclusive)
• **stop**: Ending index (exclusive)
• **step**: Step size (optional)

You can access individual characters in a string using **indices**. Python uses **zero-based indexing**.

```
s = "Python"

# Positive indices
```

```python
print(s[0])  # Output: 'P'
print(s[2])  # Output: 't'

# Negative indices
print(s[-1])  # Output: 'n'
print(s[-3])  # Output: 'h'

# Slicing
print(s[1:4])   # Output: 'yth'
print(s[:3])    # Output: 'Pyt'
print(s[3:])    # Output: 'hon'
print(s[-4:-1]) # Output: 'tho'
```

**String Concatenation and Repetition**

**Concatenation** combines two or more strings, while **repetition** repeats a string multiple times.

```python
# Concatenation
s1 = "Hello"
s2 = "World"
s3 = s1 + ", " + s2 + "!"
print(s3)  # Output: "Hello, World!"

# Repetition
s4 = "Echo! " * 3
print(s4)  # Output: "Echo! Echo! Echo! "
```

## *Common String Methods*

Python provides a rich set of **string methods** to perform various operations on strings. Below are some of the most useful and commonly used string methods.

### and

Convert a string to uppercase or lowercase.

```
s = "Hello, World!"

print(s.upper())  # Output: "HELLO, WORLD!"
print(s.lower())  # Output: "hello, world!"
```

### , , and

Remove whitespace or specified characters from the beginning and/or end of a string.

```
s = "   Hello, World!   "

# Remove leading and trailing whitespace
```

print(s.strip())   # Output: "Hello, World!"

# Remove leading whitespace
print(s.lstrip())  # Output: "Hello, World!   "

# Remove trailing whitespace
print(s.rstrip())  # Output: "   Hello, World!"

Removing Specific Characters:
s = "###Hello, World!###"

# Remove leading and trailing '#'
print(s.strip('#'))   # Output: "Hello, World!"

and

`.split()` divides a string into a list based on a specified separator. `.join()` combines elements of an iterable into a single string with a specified separator.

# Using split()
s = "apple,banana,cherry"
fruits = s.split(',')  # Split by comma
print(fruits)  # Output: ['apple', 'banana', 'cherry']

# Using join()

```
separator = "-_-"
joined = separator.join(fruits)
print(joined)  # Output: "apple-_-banana-_-cherry"
```

**Default Separator:**

• If no separator is specified, `.split()` uses any whitespace as the default separator.

```
s = "Hello World! Welcome to Python."
words = s.split()  # Split by whitespace
print(words)  # Output: ['Hello', 'World!', 'Welcome', 'to', 'Python.']
```

## .replace()

**Replace occurrences of a substring with another substring**.

```
s = "Hello, World!"
s_new = s.replace("World", "Python")
print(s_new)  # Output: "Hello, Python!"
```

**Replacing Multiple Occurrences:**

```
s = "banana"
s_new = s.replace("a", "o")
print(s_new)  # Output: "bonono"
```

**Limiting Replacements:**

The `.replace()` method can take an optional third argument specifying the maximum number of replacements.

```
s = "apple, apple, apple"
s_new = s.replace("apple", "orange", 2)
print(s_new)  # Output: "orange, orange, apple"
```

## and

`.find()` and `.index()` are used to locate the position of a substring within a string. Both return the lowest index where the substring is found.

- `.find()` returns `-1` if the substring is not found.
- `.index()` raises a `ValueError` if the substring is not found.

```
s = "Hello, World!"

# Using find()
position = s.find("World")
print(position)  # Output: 7

# Using index()
position = s.index("World")
print(position)  # Output: 7
```

```
# Substring not found
print(s.find("Python"))  # Output: -1
# print(s.index("Python"))  # Raises ValueError
```

and

**Check if a string starts or ends with a specified substring.**

```
s = "Hello, World!"
print(s.startswith("Hello"))  # Output: True
print(s.endswith("World!"))   # Output: True
print(s.startswith("World"))  # Output: False
```

**Case Sensitivity:** These methods are case-sensitive.

```
print(s.startswith("hello"))  # Output: False
```

Count the number of occurrences of a substring within a string.

```
s = "banana"
count = s.count("a")
print(count)  # Output: 3

# Counting overlapping substrings
s = "aaaaa"
count = s.count("aa")
```

print(count)  # Output: 2  # 'aa' overlaps, counted separately

## , , and

`.capitalize()` capitalizes the first character of the string and lowers the rest.

s = "hello, world!"

print(s.capitalize())  # Output: "Hello, world!"

`.title()` capitalizes the first character of each word

s = "hello, world!"

print(s.title())  # Output: "Hello, World!"

`.swapcase()` swaps the case of each character.

s = "Hello, World!"

print(s.swapcase())  # Output: "hELLO, wORLD!"

## `.()` and f-strings

### `.format()`

Use placeholders `{}` within the string and provide values via `.format()`.

name = "Alice"

age = 25

s = "My name is {} and I am {} years old.".format(name, age)

print(s)  # Output: "My name is Alice and I am 25 years old."

**Specifying Order and Keys:**

**Order**
s = "My name is {0} and I am {1} years old. {0} loves Python.".format(name, age)
print(s)  # Output: "My name is Alice and I am 25 years old. Alice loves Python."

**Keys:**
s = "My name is {name} and I am {age} years old.".format(name="Bob", age=30)print(s)  # Output: "My name is Bob and I am 30 years old

**f-strings (Formatted String Literals)**
Introduced in Python 3.6, f-strings offer a more concise and readable way to embed expressions inside string literals.
name = "Charlie"
age = 23
s = f"My name is {name} and I am {age} years old."
print(s)  # Output: "My name is Charlie and I am 23 years old."

**Including Expressions:**
s = f"Next year, I will be {age + 1} years old."
print(s)  # Output: "Next year, I will be 24 years old."

**Formatting Numbers:**

```
pi = 3.141592653589793
s = f"Pi rounded to 2 decimal places is {pi:.2f}."
print(s)  # Output: "Pi rounded to 2 decimal places is 3.14."
```

Using Methods Inside f-strings:

```
name = "david"
s = f"My name is {name.capitalize()}."
print(s)  # Output: "My name is David."
```

, , , and More

```
s1 = "HelloWorld"
print(s1.isalpha())  # Output: True  # All characters are alphabetic

s2 = "12345"
print(s2.isdigit())  # Output: True  # All characters are digits

s3 = "Hello123"
print(s3.isalnum())  # Output: True  # All characters are alphanumeric

s4 = "Hello World!"
print(s4.isalnum())  # Output: False  # Contains space and punctuation
```

```python
# Other similar methods:
# .isspace(), .islower(), .isupper(), .istitle(), etc.

s5 = "   "
print(s5.isspace())  # Output: True

s6 = "hello"
print(s6.islower())  # Output: True

s7 = "HELLO"
print(s7.isupper())  # Output: True

s8 = "Hello"
print(s8.istitle())  # Output: True
```

# *Exercise*

## Example 1: Basic String Manipulation

```python
s = "  Python Programming  "

# Remove leading and trailing whitespace
s_clean = s.strip()
print(s_clean)  # Output: "Python Programming"
```

```
# Convert to uppercase
s_upper = s_clean.upper()
print(s_upper)  # Output: "PYTHON PROGRAMMING"

# Replace "Programming" with "Language"
s_replaced = s_clean.replace("Programming", "Language")
print(s_replaced)  # Output: "Python Language"

# Split into words
words = s_clean.split()
print(words)  # Output: ['Python', 'Programming']

# Join words with a hyphen
s_joined = '-'.join(words)
print(s_joined)  # Output: "Python-Programming"
```

Example 2: Using  and f-strings

```
name = "Diana"
age = 27

# Using .format()
s1 = "My name is {} and I am {} years old.".format(name,
age)
print(s1)  # Output: "My name is Diana and I am 27 years
old."
```

# Using f-strings
s2 = f"My name is {name} and I am {age} years old."
print(s2)  # Output: "My name is Diana and I am 27 years old."

# Including expressions in f-strings
s3 = f"In five years, I will be {age + 5} years old."
print(s3)  # Output: "In five years, I will be 32 years old."

Example 2: Using  and f-strings

name = "Diana"
age = 27

# Using .format()
s1 = "My name is {} and I am {} years old.".format(name, age)
print(s1)  # Output: "My name is Diana and I am 27 years old."

# Using f-strings
s2 = f"My name is {name} and I am {age} years old."
print(s2)  # Output: "My name is Diana and I am 27 years old."

# Including expressions in f-strings
s3 = f"In five years, I will be {age + 5} years old."

```python
print(s3)  # Output: "In five years, I will be 32 years old."
```

Example 3: Checking String Content

```python
s = "Hello123"

# Check if all characters are alphanumeric
print(s.isalnum())  # Output: True

# Check if all characters are alphabetic
print(s.isalpha())  # Output: False

# Check if all characters are digits
print(s.isdigit())  # Output: False

# Check if string starts with "He"
print(s.startswith("He"))  # Output: True

# Check if string ends with "123"
print(s.endswith("123"))  # Output: True
```

## Exercise 1: Palindrome Checker

**Task:** Write a program that checks if a given string is a palindrome (reads the same backward as forward).

```python
def is_palindrome(s):
    # Remove spaces and convert to lowercase
```

```python
    s_clean = ''.join(s.split()).lower()
    return s_clean == s_clean[::-1]

# Test cases
test_strings = ["Racecar", "Hello", "A man a plan a canal
Panama", "Python"]

for string in test_strings:
    if is_palindrome(string):
        print(f"'{string}' is a palindrome.")
    else:
        print(f"'{string}' is not a palindrome.")
```

**Expected Output:**

```
Expected Output:
'Racecar' is a palindrome.
'Hello' is not a palindrome.
'A man a plan a canal Panama' is a palindrome.
'Python' is not a palindrome.
```

## Exercise 2: Sentence Capitalization

**Task:** Given a sentence, capitalize the first letter of each word.

```python
sentence = "python programming is fun!"
```

```python
# Using .title()
capitalized = sentence.title()
print(capitalized)  # Output: "Python Programming Is Fun!"

# Using .split() and .join()
words = sentence.split()
capitalized_words = [word.capitalize() for word in words]
capitalized = ' '.join(capitalized_words)
print(capitalized)  # Output: "Python Programming Is Fun!"
```

## Exercise 3: Extracting Vowels

**Task:** Extract all vowels from a given string and return them as a list.

```python
def extract_vowels(s):
    vowels = 'aeiouAEIOU'
    return [char for char in s if char in vowels]

s = "Hello, World!"
vowels_list = extract_vowels(s)
print(vowels_list)  # Output: ['e', 'o', 'o']
```

## Exercise 4: Reversing Words in a Sentence

**Task:** Reverse the order of words in a given sentence.

```
sentence = "Python is a powerful programming language."

# Split the sentence into words
words = sentence.split()

# Reverse the list of words
reversed_words = words[::-1]

# Join the reversed words back into a string
reversed_sentence = ' '.join(reversed_words)
print(reversed_sentence)  # Output: "language.
programming powerful a is Python"
```

## Exercise 5: Counting Specific Characters

```
def count_e(s):
    return s.lower().count('e')

s = "Eleven elephants entered the estate."
count = count_e(s)
print(f"The letter 'e' appears {count} times.")  # Output: "The
letter 'e' appears 7 times."
```

## **Exercise 5: Counting Specific Characters**

**Task:** Count the number of times the letter 'e' appears in a string, case-insensitive

```
def count_e(s):
    return s.lower().count('e')

s = "Eleven elephants entered the estate."
count = count_e(s)
print(f"The letter 'e' appears {count} times.")  # Output: "The letter 'e' appears 7 times."
```

# *Tuples*

## Introduction to Tuples

A **tuple** is an ordered, immutable collection of items in Python. Unlike lists, tuples cannot be modified after their creation, making them suitable for storing data that should remain constant throughout the program.

**Key Characteristics of Tuples:**

- **Ordered:** Elements have a defined order, and that order will not change.
- **Immutable:** Once a tuple is created, its elements cannot be altered, added, or removed.
- **Heterogeneous:** Tuples can contain elements of different

data types.
• **Indexed:** Elements can be accessed using indices, starting from 0.
• **Hashable:** Tuples can be used as keys in dictionaries if all their elements are hashable.

**Common Use Cases:**
• Returning multiple values from a function.
• Storing related but different pieces of data.
• Using as keys in dictionaries.
• Ensuring data integrity by preventing modifications.

# *Tuple Basics*

Tuples can be created in several ways:

**Using Parentheses** ()

```
# Empty tuple
empty_tuple = ()

# Tuple with multiple elements
fruits = ("apple", "banana", "cherry")
print(fruits)  # Output: ('apple', 'banana', 'cherry')
```

**Without Parentheses (Tuple Packing)**

```
# Tuple packing
colors = "red", "green", "blue"
print(colors)  # Output: ('red', 'green', 'blue')
```

**Using the `tuple()` Construct**or

```
# From a list
numbers = tuple([1, 2, 3, 4])
print(numbers)  # Output: (1, 2, 3, 4)

# From a string
letters = tuple("Python")
print(letters)  # Output: ('P', 'y', 't', 'h', 'o', 'n')
```

**Note:** Parentheses are optional when defining a tuple; however, they enhance readability, especially in complex expressions.

**Tuple Immutability**

Tuples are **immutable**, meaning that once a tuple is created, its elements cannot be changed, added, or removed.

```
s = (1, 2, 3)
# Attempting to modify an element
```

# s[0] = 10  # This will raise a TypeError

# Attempting to add an element
# s.append(4)  # AttributeError: 'tuple' object has no attribute 'append'

# Attempting to remove an element
# del s[1]  # TypeError: 'tuple' object doesn't support item deletion

**Why Use Immutable Tuples?**

- **Data Integrity:** Ensures that the data remains constant throughout the program.
- **Hashable:** Allows tuples to be used as keys in dictionaries and elements in sets.
- **Performance:** Slightly faster than lists due to immutability.

## Tuple Indexing and Slicing

Like lists, tuples support **indexing** and **slicing** to access their elements.

Slicing Syntax:

- **start:** Starting index (inclusive).
- **stop:** Ending index (exclusive).
- **step:** Step size (optional).

```
s = ("a", "b", "c", "d", "e")

# Indexing
print(s[0])   # Output: 'a'
print(s[-1])  # Output: 'e'

# Slicing
print(s[1:4])   # Output: ('b', 'c', 'd')
print(s[:3])    # Output: ('a', 'b', 'c')
print(s[2:])    # Output: ('c', 'd', 'e')
print(s[::2])   # Output: ('a', 'c', 'e')
```

## Tuple Concatenation and Repetition

Tuples support **concatenation** using the + operator and **repetition** using the * operator.

```
t1 = (1, 2, 3)
t2 = (4, 5)

# Concatenation
t3 = t1 + t2
print(t3)  # Output: (1, 2, 3, 4, 5)

# Repetition
```

```
t4 = ("repeat",) * 3
print(t4)  # Output: ('repeat', 'repeat', 'repeat')
```

**Important Note on Repetition:**

• To create a single-element tuple, include a comma:

```
single = ("single",)
print(single)  # Output: ('single',)
```

Without the comma, it's just a string within parentheses:

```
not_a_tuple = ("single")
print(not_a_tuple)  # Output: 'single'
```

## Single-Element Tuples

Creating a tuple with a single element requires a trailing comma to differentiate it from a regular parenthesis-enclosed expression.

```
# Single-element tuple
singleton = (42,)
print(singleton)  # Output: (42,)

# Not a tuple
not_tuple = (42)
print(not_tuple)  # Output: 42
```

**Why the Comma?**

• The comma is the defining characteristic of a tuple, not the parentheses. Parentheses are used for grouping, while commas indicate tuple elements

# *Tuple Operations*

## Tuple Unpacking

**Tuple unpacking** allows you to assign the elements of a tuple to individual variables in a single statement.

```
# Unpacking a tuple
person = ("Alice", 30, "Engineer")

name, age, profession = person
print(name)      # Output: Alice
print(age)       # Output: 30
print(profession) # Output: Engineer
```

**Error Example (Mismatch in Elements):**

```
data = (1, 2, 3)
# a, b = data  # ValueError: too many values to unpack
(expected 2)
```

**Handling Variable Number of Elements:**

```
# Using * to capture remaining elements
data = (1, 2, 3, 4, 5)
a, b, *c = data
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: [3, 4, 5]
```

## Packing Tuples

**Tuple packing** is the process of combining multiple values into a single tuple without explicitly using parentheses.

```
# Tuple packing
packed = "x", "y", "z"
print(packed)  # Output: ('x', 'y', 'z')

# Equivalent to
packed = ("x", "y", "z")
print(packed)  # Output: ('x', 'y', 'z')
```

**Use Cases:**

- Returning multiple values from a function.
- Passing multiple values as a single entity.

## Swapping Variables

Tuples make it easy to swap the values of two variables without needing a temporary variable.

```
a = 10
b = 20

# Swapping using tuple unpacking
a, b = b, a

print(a)  # Output: 20
print(b)  # Output: 10
```

**Explanation:**

- The right-hand side `(b, a)` creates a tuple `(20, 10)`.
- The left-hand side `a, b` unpacks the tuple, assigning `20` to `a` and `10` to `b`.

## *Common Tuple Methods*

Tuples have a minimal set of built-in methods due to their immutability. Here are the most commonly used ones:

### `.count()`

Counts the number of occurrences of a specified value in the tuple.

```
t = (1, 2, 3, 2, 4, 2, 5)

count_2 = t.count(2)
print(count_2)  # Output: 3

count_6 = t.count(6)
print(count_6)  # Output: 0
```

## .index()

Returns the index of the first occurrence of a specified value. Raises a `ValueError` if the value is not found.

```
t = ("apple", "banana", "cherry", "banana")

index_banana = t.index("banana")
print(index_banana)  # Output: 1

# Finding index with start and end parameters
index_banana_second = t.index("banana", 2)
print(index_banana_second)  # Output: 3

# Value not in tuple
# t.index("orange")  # Raises ValueError: tuple.index(x): x not in tuple
```

**Handling ValueError:**

```
t = (1, 2, 3)
```

```
try:
    t.index(4)
except ValueError:
    print("Value not found in tuple.")  # Output: Value not found in tuple.
```

## Nested Tuples

Tuples can contain other tuples (or other data structures) as elements, allowing the creation of multi-dimensional tuples.

# *Advanced Tuple Concepts*


## Nested Tuples

Tuples can contain other tuples (or other data structures) as elements, allowing the creation of multi-dimensional tuples.


```
nested = ((1, 2), (3, 4), (5, 6))
```

```
print(nested)         # Output: ((1, 2), (3, 4), (5, 6))
print(nested[0])      # Output: (1, 2)
print(nested[0][1])   # Output: 2
```

## Named Tuples

**Named Tuples** provide a way to define simple classes with named fields, making tuples more readable and accessible.

**Advantages:**

- Access elements by name instead of index.
- Improves code readability.
- Immutable like regular tuples.

**Using the `collections` module:**

```
from collections import namedtuple

# Define a named tuple type
Person = namedtuple('Person', ['name', 'age', 'profession'])

# Create instances
alice = Person(name="Alice", age=30, profession="Engineer")
bob = Person("Bob", 25, "Designer")

print(alice)      # Output: Person(name='Alice', age=30, profession='Engineer')
print(bob.name)   # Output: Bob
```

print(bob.age)    # Output: 25

**Accessing Elements:**
print(alice.name)      # Output: Alice
print(alice[1])        # Output: 30 (index-based access is also allowed)

**Extending Named Tuples:**
Named tuples can be extended by defining new named tuple types or by using inheritance.

# Define a new named tuple type that extends Person
Employee = namedtuple('Employee', Person._fields + ('employee_id',))

# Create an instance
charlie = Employee(name="Charlie", age=28, profession="Developer", employee_id=1001)
print(charlie)          # Output: Employee(name='Charlie', age=28, profession='Developer', employee_id=1001)
print(charlie.employee_id)  # Output: 1001

# ***Tuple Comprehensions***

Python does **not** support tuple comprehensions in the same way it does list comprehensions. Instead, generator expressions are used for similar purposes, as tuples require all elements to be generated upfront.

However, you can create a tuple from a generator expression using the `tuple()` constructor.
# Attempting a tuple comprehension-like syntax (Not valid)
# t = (x for x in range(5))  # This creates a generator, not a tuple

# Correct way to create a tuple from a generator expression
t = tuple(x for x in range(5))
print(t)  # Output: (0, 1, 2, 3, 4)

## ***Tuples vs Lists***

### 6. Tuples vs. Lists

Understanding the differences between tuples and lists is crucial for choosing the appropriate data structure based on the use case.

| Feature | Tuples | Lists |
|---|---|---|
| **Syntax** | Parentheses `()` or no parentheses | Square brackets `[]` |
| **Mutability** | Immutable | Mutable |
| **Use Cases** | Fixed collections, dictionary keys | Dynamic collections, collections needing modification |
| **Performance** | Slightly faster due to immutability | Slower for certain operations |
| **Methods Available** | `.count()`, `.index()` | Numerous methods like `.append()`, `.remove()`, `.pop()`, etc. |
| **Memory Consumption** | Generally uses less memory | Generally uses more memory |
| **Hashability** | Hashable if all elements are hashable | Not hashable |

## When to Use Tuples

- **Fixed Data:** When the collection of data should not change.
- **Dictionary Keys:** Tuples can be used as keys in dictionaries, while lists cannot.
- **Data Integrity:** Ensuring that data remains constant throughout the program.
- **Performance:** Slightly faster access and iteration.

### When to Use Lists

◇ **Dynamic Data:** When the collection needs to be modified (add, remove, change elements).
◇ **Homogeneous Data:** Often used for collections of similar items.
◇ **Advanced Operations:** When you need to perform operations like sorting, reversing, etc.

## *Examples and Exercises*

### Example 1: Returning Multiple Values from a Function

Tuples are commonly used to return multiple values from a function.

```
def get_user_info():
    name = "Eve"
    age = 28
    profession = "Data Scientist"
    return name, age, profession  # Returns a tuple

user_info = get_user_info()
```

```python
print(user_info)  # Output: ('Eve', 28, 'Data Scientist')

# Unpacking the tuple
name, age, profession = get_user_info()
print(name)       # Output: Eve
print(age)        # Output: 28
print(profession)  # Output: Data Scientist
```

## Example 2: Using Named Tuples for Structured Data

Named tuples enhance the readability and usability of tuples by allowing access via named fields.

```python
from collections import namedtuple

# Define the named tuple
Point = namedtuple('Point', ['x', 'y'])

# Create instances
p1 = Point(10, 20)
p2 = Point(x=15, y=25)

print(p1)         # Output: Point(x=10, y=20)
print(p2.x, p2.y)  # Output: 15 25
```

## Exercise 1: Finding Unique Elements

**Task:** Given a tuple of numbers, return a new tuple containing only the unique elements.

**List:** `(1, 2, 2, 3, 4, 4, 5)`

**Expected Output:** `(1, 2, 3, 4, 5)`

```
def unique_elements(t):
    return tuple(set(t))

numbers = (1, 2, 2, 3, 4, 4, 5)
unique = unique_elements(numbers)
print(unique)  # Output: (1, 2, 3, 4, 5)
```

## Exercise 2: Merging Two Tuples

**Task:** Merge two tuples into a single tuple.

**Tuples:**

- `t1 = (1, 2, 3)`
- `t2 = (4, 5, 6)`

**Expected Output:** `(1, 2, 3, 4, 5, 6)`

```
t1 = (1, 2, 3)
t2 = (4, 5, 6)
```

```
merged = t1 + t2
print(merged)  # Output: (1, 2, 3, 4, 5, 6)
```

## Exercise 3: Reversing a Tuple

**Task:** Reverse the elements of a tuple.
**Tuple:** `(1, 2, 3, 4, 5)`
**Expected Output:** `(5, 4, 3, 2, 1)`

```
t = (1, 2, 3, 4, 5)
reversed_t = t[::-1]
print(reversed_t)  # Output: (5, 4, 3, 2, 1)
```

## Exercise 4: Checking for an Element in a Tuple

**Task:** Write a function that checks if a given element exists in a tuple.
**Tuple:** `("apple", "banana", "cherry")`
**Function:** `def contains(t, element):`

```
t = ("apple", "banana", "cherry")

def contains(t, element):
    return element in t

print(contains(t, "banana"))  # Output: True
print(contains(t, "grape"))   # Output: False
```

### Exercise 5: Counting Vowels in a Tuple of Strings

**Task:** Given a tuple of strings, count the total number of vowels present.

**Tuple:** `("hello", "world", "python", "programming")`

**Expected Output:** `7`

**Solution:**

```
def count_vowels(t):
    vowels = 'aeiouAEIOU'
    return sum(1 for word in t for char in word if char in vowels)

words = ("hello", "world", "python", "programming")
total_vowels = count_vowels(words)
print(total_vowels)  # Output: 7
```

# *Sets*

## Introduction to Sets

A **set** is an unordered collection of unique elements in Python. Sets are mutable, meaning you can add or remove elements after creation. However, the elements contained within a set must be **hashable** (immutable types like numbers, strings, and tuples).

**Key Characteristics of Sets:**

- **Unordered:** Sets do not maintain any order. The elements have no index positions.
- **Unique:** All elements in a set are unique; duplicates are automatically removed.
- **Mutable:** You can add or remove elements from a set.
- **Iterable:** You can loop through the elements in a set.

**Common Use Cases:**

◇ **Membership Testing:** Quickly check if an element exists in a collection.

◇ **Removing Duplicates:** Eliminate duplicate entries from a list.

◇ **Set Operations:** Perform mathematical set operations like union, intersection, and difference.

# *Set Basics*

## Creating Sets

There are several ways to create sets in Python:

**Using Curly Braces {}:**

```
# Creating a set with multiple elements
fruits = {"apple", "banana", "cherry"}
print(fruits)  # Output: {'banana', 'cherry', 'apple'}

# Creating an empty set (Note: `{}` creates an empty dictionary)
empty_set = set()
print(empty_set)  # Output: set()
```

**Using the set() Constructor:**

```
# From a list
numbers = set([1, 2, 3, 4, 5])
print(numbers)  # Output: {1, 2, 3, 4, 5}

# From a string
letters = set("Python")
print(letters)  # Output: {'n', 'h', 'y', 'P', 't', 'o'}

# From a tuple
t = set((1, 2, 3))
print(t)  # Output: {1, 2, 3}
```

**Using Set Comprehensions:**

```
# Creating a set of squares from 0 to 9
squares = {x**2 for x in range(10)}
```

print(squares)  # Output: {0, 1, 64, 4, 9, 16, 25, 36, 49, 81}

**Important Notes:**
**Uniqueness:** When creating a set, duplicate elements are automatically removed
duplicates = {1, 2, 2, 3, 4, 4, 5}
print(duplicates)  # Output: {1, 2, 3, 4, 5}

**Immutable Elements:** Sets cannot contain mutable elements like lists or dictionaries.
# This will raise a TypeError
invalid_set = {1, 2, [3, 4]}
# TypeError: unhashable type: 'list'

**Set Immutability**
While **sets themselves are mutable** (you can add or remove elements), the **elements within a set must be immutable** (hashable).

Mutable Elements (Not Allowed):
# Attempting to add a list to a set
s = {1, 2, 3}
s.add([4, 5])  # Raises TypeError: unhashable type: 'list'

Immutable Elements (Allowed):

```
# Adding a tuple to a set
s = {1, 2, 3}
s.add((4, 5))
print(s)  # Output: {1, 2, 3, (4, 5)}
```

## Set Indexing and Slicing

**Sets are unordered**, which means they do not support indexing, slicing, or other sequence-like behavior

```
s = {"apple", "banana", "cherry"}

# Attempting to access an element by index
# print(s[0])  # Raises TypeError: 'set' object is not subscriptable

# Iterating through a set
for fruit in s:
    print(fruit)
```

**Output:**

```
banana
cherry
apple
```

**Note:** The order of elements when iterating over a set is not guaranteed and may vary.

**Set Concatenation and Repetition**

Sets do not support concatenation (`+`) or repetition (`*`) operations like lists and strings because they are unordered and contain unique element

```
s1 = {"a", "b", "c"}
s2 = {"d", "e", "f"}

# Concatenation (Not Supported)
# combined = s1 + s2  # Raises TypeError: unsupported operand type(s) for +: 'set' and 'set'

# Repetition (Not Supported)
# repeated = s1 * 2  # Raises TypeError: unsupported operand type(s) for *: 'set' and 'int'
```

**Alternative for Union:**

Use the `union()` method or the `|` operator to combine sets.

```
# Using union()
combined = s1.union(s2)
print(combined)  # Output: {'a', 'b', 'c', 'd', 'e', 'f'}

# Using | operator
combined = s1 | s2
```

```
print(combined)  # Output: {'a', 'b', 'c', 'd', 'e', 'f'}
```

**Single-Element Sets**

To create a set with a single element, include a trailing comma inside the curly braces.

```
# Single-element set
singleton = {"apple"}
print(singleton)  # Output: {'apple'}

# However, {} is an empty dictionary, not a set
empty = {}
print(type(empty))  # Output: <class 'dict'>

# Correct way for empty set
empty_set = set()
print(type(empty_set))  # Output: <class 'set'>
```

**Note:** `{}` creates an empty dictionary, not a set. Use `set()` to create an empty set.

# *Set Operations*

Sets support various mathematical operations, allowing for efficient handling of unique elements.

## Mathematical Set Operations

### Union (`|` or `.union()`)

Combines elements from two or more sets, removing duplicates.

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}

# Using | operator
union_set = s1 | s2
print(union_set)  # Output: {1, 2, 3, 4, 5}

# Using .union() method
union_set = s1.union(s2)
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

**Intersection (`&` or `.intersection()`)**

Returns elements common to both sets.

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
```

# Using & operator
intersection_set = s1 & s2
print(intersection_set)  # Output: {3}

# Using .intersection() method
intersection_set = s1.intersection(s2)
print(intersection_set)  # Output: {3}

**Difference (`-` or `.difference()`)**

Returns elements present in the first set but not in the second.

s1 = {1, 2, 3}
s2 = {3, 4, 5}

# Using - operator
difference_set = s1 - s2
print(difference_set)  # Output: {1, 2}

# Using .difference() method
difference_set = s1.difference(s2)
print(difference_set)  # Output: {1, 2}

**Symmetric Difference (`^` or `.symmetric_difference()`)**

Returns elements present in either set but not in both.

s1 = {1, 2, 3}

s2 = {3, 4, 5}

# Using ^ operator
sym_diff_set = s1 ^ s2
print(sym_diff_set)  # Output: {1, 2, 4, 5}

# Using .symmetric_difference() method
sym_diff_set = s1.symmetric_difference(s2)
print(sym_diff_set)  # Output: {1, 2, 4, 5}

## Subset and Superset

### Subset (`<=` or `.issubset()`)

Checks if all elements of one set are present in another.

s1 = {1, 2}
s2 = {1, 2, 3}

# Using <= operator
is_subset = s1 <= s2
print(is_subset)  # Output: True

# Using .issubset() method
is_subset = s1.issubset(s2)
print(is_subset)  # Output: True

### Superset (`>=` or `.issuperset()`)

**Description:** Checks if a set contains all elements of another set.

```
s1 = {1, 2, 3}
s2 = {1, 2}

# Using >= operator
is_superset = s1 >= s2
print(is_superset)  # Output: True

# Using .issuperset() method
is_superset = s1.issuperset(s2)
print(is_superset)  # Output: True
```

## Disjoint Sets

**Description:** Determines if two sets have no elements in common

```
s1 = {1, 2, 3}
s2 = {4, 5, 6}
s3 = {3, 4}

# Using .isdisjoint() method
print(s1.isdisjoint(s2))  # Output: True
print(s1.isdisjoint(s3))  # Output: False
```

# *Common Set Methods*

While sets have fewer methods compared to lists, they still offer a range of functionalities to manipulate and interact with their elements.

### add()

**Description:** Adds a single element to the set.

```
s = {1, 2, 3}
s.add(4)
print(s)  # Output: {1, 2, 3, 4}

# Adding an element that already exists has no effect
s.add(2)
print(s)  # Output: {1, 2, 3, 4}
```

**update()**

**Description:** Adds multiple elements to the set from an iterable (like lists, tuples, or other sets).

```
s = {1, 2, 3}
s.update([4, 5], {6, 7}, (8, 9))
print(s)  # Output: {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

**remove()**

**Description:** Removes a specified element from the set. Raises a `KeyError` if the element is not found.

```
s = {1, 2, 3, 4}
s.remove(3)
print(s)  # Output: {1, 2, 4}

# Attempting to remove a non-existent element
# s.remove(5)  # Raises KeyError: 5
```

**`discard()`**

**Description:** Removes a specified element from the set if it exists. Does **not** raise an error if the element is not found.

```
s = {1, 2, 3, 4}
s.discard(3)
print(s)  # Output: {1, 2, 4}

# Discarding a non-existent element does nothing
s.discard(5)
print(s)  # Output: {1, 2, 4}
```

### pop()

**Description:** Removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.

```
s = {1, 2, 3}
element = s.pop()
print(element)  # Output: 1 (could be any element)
print(s)        # Output: {2, 3} (assuming 1 was popped)
```

**Note:** Since sets are unordered, pop() does not guarantee which element will be removed.

### clear()

**Description:** Removes all elements from the set, resulting in an empty set

```
s = {1, 2, 3}
s.clear()
print(s)  # Output: set()
```

### copy()

**Description:** Returns a shallow copy of the set.

```
s1 = {1, 2, 3}
s2 = s1.copy()
print(s2)  # Output: {1, 2, 3}

# Modifying s2 does not affect s1
```

s2.add(4)
print(s1)  # Output: {1, 2, 3}
print(s2)  # Output: {1, 2, 3, 4}

## issubset()

**Description:** Checks if the set is a subset of another set.

s1 = {1, 2}
s2 = {1, 2, 3}
print(s1.issubset(s2))  # Output: True

s3 = {2, 4}
print(s3.issubset(s2))  # Output: False

## issuperset()

**Description:** Checks if the set is a superset of another set.

s1 = {1, 2, 3}
s2 = {1, 2}
print(s1.issuperset(s2))  # Output: True

s3 = {2, 4}
print(s1.issuperset(s3))  # Output: False

## isdisjoint()

**Description:** Checks if two sets have no elements in common

```
s1 = {1, 2, 3}
s2 = {4, 5, 6}
s3 = {3, 4}

print(s1.isdisjoint(s2))  # Output: True
print(s1.isdisjoint(s3))  # Output: False
```

# ***Advanced Set Concepts***

## Frozen Sets

A **frozen set** is an immutable version of a set. Once created, you cannot add or remove elements from a frozen set. However, you can perform set operations that return new frozen sets.

**Creating a Frozen Set:**

```
frozen = frozenset([1, 2, 3, 4])
print(frozen)  # Output: frozenset({1, 2, 3, 4})
```

**Key Characteristics:**

• **Immutable:** Cannot modify after creation.
• **Hashable:** Can be used as keys in dictionaries or elements in other sets.

Usage Example:

```
# Using frozenset as dictionary keys
d = {}
key = frozenset(["apple", "banana"])
d[key] = "Fruit Basket"
print(d)  # Output: {frozenset({'apple', 'banana'}): 'Fruit Basket'}
```

## Set Comprehensions

**Description:** Similar to list comprehensions, set comprehensions provide a concise way to create sets.

**Syntax:**

```
{expression for item in iterable if condition}
```

Example: Creating a Set of Squares:

```
squares = {x**2 for x in range(10)}
print(squares)  # Output: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

**Example: Extracting Vowels from a String:**

```
sentence = "Hello, World!"
vowels = {char.lower() for char in sentence if char.lower() in 'aeiou'}
print(vowels)  # Output: {'e', 'o'}
```

# *Sets vs. Other Data Structures*

Understanding the differences between sets and other Python data structures like lists, tuples, and dictionaries helps in choosing the right tool for your specific needs.

**Sets vs. Lists**

| Feature | Sets | Lists |
|---|---|---|
| **Ordering** | Unordered | Ordered |
| **Duplicates** | No duplicates allowed | Duplicates allowed |
| **Mutability** | Mutable | Mutable |
| **Indexing** | Not supported | Supported |
| **Performance** | Faster for membership testing | Slower for membership testing |
| **Use Cases** | Removing duplicates, set operations | Maintaining ordered collections, allowing duplicates |

**Example: Removing Duplicates from a List Using a Set**

```
fruits = ["apple", "banana", "cherry", "apple", "banana"]
unique_fruits = set(fruits)
print(unique_fruits)  # Output: {'banana', 'cherry', 'apple'}
```

## Sets vs. Tuples

| Feature | Sets | Tuples |
|---|---|---|
| **Ordering** | Unordered | Ordered |
| **Duplicates** | No duplicates allowed | Duplicates allowed |
| **Mutability** | Mutable | Immutable |
| **Indexing** | Not supported | Supported |
| **Use Cases** | Membership testing, uniqueness | Fixed collections, data integrity |

**Example: Using a Tuple as a Dictionary Key (Possible) vs. Using a List as a Key (Not Possible)**

```
# Using a tuple as a dictionary key
t = (1, 2, 3)
d = {t: "Value"}
print(d)  # Output: {(1, 2, 3): 'Value'}

# Using a list as a dictionary key (Raises TypeError)
# l = [1, 2, 3]
# d = {l: "Value"}  # Raises TypeError: unhashable type: 'list'
```

### Sets vs. Dictionaries

| Feature | Sets | Dictionaries |
|---|---|---|
| **Purpose** | Store unique elements | Store key-value pairs |
| **Structure** | Single elements | Key-value pairs |
| **Ordering** | Unordered | Ordered (as of Python 3.7+) |
| **Mutability** | Mutable | Mutable |
| **Duplicates** | No duplicates allowed | Keys must be unique; values can be duplicates |
| **Use Cases** | Membership testing, unique collections | Fast lookup, mapping relationships |

Example: Using a Set vs. Dictionary

```
# Using a set for membership testing
fruits = {"apple", "banana", "cherry"}
print("banana" in fruits)  # Output: True

# Using a dictionary for mapping
fruit_prices = {"apple": 0.99, "banana": 0.59, "cherry": 2.99}
print(fruit_prices["banana"])  # Output: 0.59
```

# ***Examples and Exercises***

## Example 1: Removing Duplicates from a List

**Task:** Given a list with duplicate elements, remove the duplicates using a set.

```
fruits = ["apple", "banana", "cherry", "apple", "banana",
"date"]
unique_fruits = set(fruits)
print(unique_fruits)  # Output: {'banana', 'date', 'cherry',
'apple'}
```

## Example 2: Membership Testing

**Task:** Check if a specific element exists in a set.

```
s = {"apple", "banana", "cherry"}
print("banana" in s)  # Output: True
print("grape" in s)   # Output: False
```

## Example 3: Set Operations

**Task:** Perform union, intersection, difference, and symmetric difference between two sets.

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}

# Union
print(s1 | s2)  # Output: {1, 2, 3, 4, 5, 6}

# Intersection
```

print(s1 & s2)  # Output: {3, 4}

# Difference
print(s1 - s2)  # Output: {1, 2}
print(s2 - s1)  # Output: {5, 6}

# Symmetric Difference
print(s1 ^ s2)  # Output: {1, 2, 5, 6}

## Exercise 1: Finding Common Elements

**Task:** Given two lists, find the common elements using sets.

**Lists:**

list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]

**Expected Output:** {4, 5}

**Solution:**

list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]

set1 = set(list1)
set2 = set(list2)

common = set1 & set2

print(common)  # Output: {4, 5}

## Exercise 2: Counting Unique Words

**Task:** Given a sentence, count the number of unique words using sets.

**Sentence:** `"Python is great and Python is fun"`

**Expected Output:** `{'Python', 'is', 'great', 'and', 'fun'}`

Solution:

```
sentence = "Python is great and Python is fun"
words = sentence.split()
unique_words = set(words)
print(unique_words)  # Output: {'is', 'Python', 'and', 'great', 'fun'}
print(len(unique_words))  # Output: 5
```

## Exercise 3: Power Set

**Task:** Generate the power set of a given set (all possible subsets).

**Set:** `{1, 2}`

**Expected Output:** `{frozenset(), frozenset({1}), frozenset({2}), frozenset({1, 2})}`

**Solution:**

```
from itertools import chain, combinations

def power_set(s):
    s = list(s)
    return set(frozenset(combo) for combo in
chain.from_iterable(combinations(s, r) for r in range(len(s)
+1)))

s = {1, 2}
ps = power_set(s)
print(ps)
# Output: {frozenset(), frozenset({1}), frozenset({2}),
frozenset({1, 2})}
```

## Exercise 4: Checking for Anagrams

**Task:** Write a function to check if two strings are anagrams using sets.

**Example:**

- `"listen"` and `"silent"` → **True**
- `"hello"` and `"world"` → **False**

**Solution:**

While sets can be used to check for the presence of the same unique characters, they **cannot** account for the frequency of each character, which is essential for

determining anagrams. Instead, it's better to use sorted strings or dictionaries.
However, to adhere to the exercise, here's how sets can partially check for anagrams

```
def are_anagrams_set(s1, s2):
    return set(s1) == set(s2)

print(are_anagrams_set("listen", "silent"))  # Output: True
print(are_anagrams_set("hello", "world"))    # Output: False

# Note: This method does not consider character counts
print(are_anagrams_set("aabb", "ab"))        # Output: True (Incorrect for anagrams)
```

**Proper Anagram Check Using Sorted Strings:**

```
def are_anagrams(s1, s2):
    return sorted(s1) == sorted(s2)

print(are_anagrams("listen", "silent"))  # Output: True
print(are_anagrams("hello", "world"))    # Output: False
print(are_anagrams("aabb", "ab"))        # Output: False
```

## Exercise 5: Removing Specific Elements

**Task:** Given a set of numbers, remove all even numbers.

**Set:** `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`
**Expected Output:** `{1, 3, 5, 7, 9}`

**Solution:**

```
numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

# Using set comprehension to filter out even numbers
odds = {x for x in numbers if x % 2 != 0}
print(odds)  # Output: {1, 3, 5, 7, 9}

# Alternatively, removing in-place
numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
for x in list(numbers):
    if x % 2 == 0:
        numbers.remove(x)
print(numbers)  # Output: {1, 3, 5, 7, 9}
```

# ***Dictionaries***

## 1. Introduction to Dictionaries

A **dictionary** in Python is an **unordered** collection of **key-value pairs**, where each key is unique and maps to a corresponding value. Dictionaries are mutable, meaning you can add, remove, or modify key-value pairs after creation.

**Key Characteristics of Dictionaries:Key Characteristics of Dictionaries:**

• **Unordered:** As of Python 3.7, dictionaries maintain the insertion order, but they are still fundamentally considered unordered collections.
• **Mutable:** You can modify dictionaries by adding, removing, or updating key-value pairs.
• **Indexed by Keys:** Unlike lists or tuples, dictionaries are indexed by keys, not by numerical indices.
• **Dynamic:** They can grow or shrink as needed.
• **Heterogeneous:** Keys and values can be of different data types.

**Common Use Cases:**

• **Mapping:** Associating unique keys with values (e.g., user IDs to user information).
• **Fast Lookup:** Retrieving data quickly using keys.
• **Data Representation:** Representing JSON-like data structures.
• **Counting and Grouping:** Tallying occurrences or grouping related data.

## *Dictionary Basics*

### Creating Dictionaries

There are multiple ways to create dictionaries in Python:

**Using Curly Braces `{}`:**

```
# Empty dictionary
empty_dict = {}
print(empty_dict)  # Output: {}

# Dictionary with key-value pairs
person = {
    "name": "Alice",
    "age": 30,
    "profession": "Engineer"
}
print(person)
# Output: {'name': 'Alice', 'age': 30, 'profession': 'Engineer'}
```

**Using the `dict()` Constructor:**

```
# From keyword arguments
car = dict(make="Toyota", model="Camry", year=2020)
print(car)
# Output: {'make': 'Toyota', 'model': 'Camry', 'year': 2020}

# From a list of tuples
fruits = dict([("apple", 2), ("banana", 3), ("cherry", 5)])
print(fruits)
# Output: {'apple': 2, 'banana': 3, 'cherry': 5}

# From a list of lists
capitals = dict([["USA", "Washington D.C."], ["France",
"Paris"], ["Japan", "Tokyo"]])
print(capitals)
# Output: {'USA': 'Washington D.C.', 'France': 'Paris', 'Japan':
'Tokyo'}
```

**Using Dictionary Comprehensions:**

```
# Creating a dictionary of squares
squares = {x: x**2 for x in range(6)}
print(squares)
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

**Important Notes:**

• **Keys Must Be Hashable:** Immutable types like strings, numbers, and tuples can be used as keys. Mutable types like lists or other dictionaries cannot.

• **Unique Keys:** Each key in a dictionary must be unique. If duplicate keys are provided, the last occurrence overwrites the previous ones.

```
duplicate_keys = {"a": 1, "b": 2, "a": 3}
print(duplicate_keys)  # Output: {'a': 3, 'b': 2}
```

## Accessing Values

You can access dictionary values using their corresponding keys:

**Using Square Brackets `[]`:**

```
person = {"name": "Alice", "age": 30, "profession":
"Engineer"}
print(person["name"])  # Output: Alice
print(person["age"])   # Output: 30
```

**Note:** Accessing a non-existent key using `[]` raises a `KeyError`.

```
print(person["salary"])  # Raises KeyError: 'salary'
```

**Using the `.get()` Method:**
print(person.get("name"))    # Output: Alice
print(person.get("salary"))   # Output: None
print(person.get("salary", 50000))  # Output: 50000 (default value)

**Advantages of `.get()`:**
• Avoids `KeyError` by returning `None` or a specified default value if the key is not found.

## Modifying Dictionaries

Dictionaries are mutable, allowing you to add, update, or remove key-value pairs.

**Adding or Updating Entries:**
person = {"name": "Alice", "age": 30}

# Adding a new key-value pair
person["profession"] = "Engineer"
print(person)
# Output: {'name': 'Alice', 'age': 30, 'profession': 'Engineer'}

# Updating an existing key
person["age"] = 31
print(person)

# Output: {'name': 'Alice', 'age': 31, 'profession': 'Engineer'}

**Removing Entries:**

**Using `del`:**

del person["age"]
print(person)
# Output: {'name': 'Alice', 'profession': 'Engineer'}

**Note:** Attempting to delete a non-existent key raises a `KeyError`.

**Using `.pop()`:**

profession = person.pop("profession")
print(profession)  # Output: Engineer
print(person)      # Output: {'name': 'Alice'}

**Using `.popitem()`:**

# Removes and returns an arbitrary key-value pair (as of Python 3.7, it removes the last inserted item)
last_item = person.popitem()
print(last_item)  # Output: ('name', 'Alice')
print(person)     # Output: {}

**Clearing the Dictionary:**

person = {"name": "Alice", "age": 30, "profession":

```python
"Engineer"}
person.clear()
print(person)  # Output: {}
```

**Dictionary Immutability**

While the **dictionary structure is mutable**, meaning you can add, remove, or change key-value pairs, the **keys themselves must be immutable** (hashable). However, the **values can be of any type**, including mutable ones like lists or other dictionaries.

```python
# Dictionary with mutable values
data = {
    "fruits": ["apple", "banana"],
    "numbers": [1, 2, 3],
    "details": {"name": "Alice", "age": 30}
}

# Modifying a mutable value
data["fruits"].append("cherry")
print(data["fruits"])  # Output: ['apple', 'banana', 'cherry']
```

**Immutable Keys:**

```
# Dictionary with mutable values
data = {
    "fruits": ["apple", "banana"],
    "numbers": [1, 2, 3],
    "details": {"name": "Alice", "age": 30}
}

# Modifying a mutable value
data["fruits"].append("cherry")
print(data["fruits"])  # Output: ['apple', 'banana', 'cherry']
```

# *Common Dictionary Methods*

**Common Dictionary Methods**
Dictionaries in Python come with a rich set of methods that allow for efficient data manipulation. Below are some of the most commonly used dictionary methods with examples.

**keys()**
**Description:** Returns a view object containing all the keys in the dictionary.

```
person = {"name": "Alice", "age": 30, "profession":
"Engineer"}
keys = person.keys()
print(keys)  # Output: dict_keys(['name', 'age', 'profession'])

# Iterating through keys
for key in person.keys():
    print(key)
# Output:
# name
# age
# profession
```

### `values()`

**Description:** Returns a view object containing all the values in the dictionary.

```
person = {"name": "Alice", "age": 30, "profession":
"Engineer"}
values = person.values()
print(values)  # Output: dict_values(['Alice', 30, 'Engineer'])

# Iterating through values
for value in person.values():
```

```
    print(value)
# Output:
# Alice
# 30
# Engineer
```

## `items()`

**Description:** Returns a view object containing all key-value pairs as tuples.

```
person = {"name": "Alice", "age": 30, "profession":
"Engineer"}
items = person.items()
print(items)  # Output: dict_items([('name', 'Alice'), ('age', 30),
('profession', 'Engineer')])

# Iterating through key-value pairs
for key, value in person.items():
    print(f"{key}: {value}")
# Output:
# name: Alice
# age: 30
# profession: Engineer
```

## `get()`

**Description:** Returns the value for a specified key if the key exists; otherwise, returns None or a specified default value.

```
person = {"name": "Alice", "age": 30}

# Existing key
name = person.get("name")
print(name)  # Output: Alice

# Non-existent key without default
salary = person.get("salary")
print(salary)  # Output: None

# Non-existent key with default
salary = person.get("salary", 50000)
print(salary)  # Output: 50000
```

### update()

**Description:** Updates the dictionary with key-value pairs from another dictionary or an iterable of key-value pairs.

```
person = {"name": "Alice", "age": 30}
additional_info = {"profession": "Engineer", "age": 31}

# Updating with another dictionary
```

```python
person.update(additional_info)
print(person)
# Output: {'name': 'Alice', 'age': 31, 'profession': 'Engineer'}

# Updating with an iterable of key-value pairs
person.update([("salary", 70000), ("city", "New York")])
print(person)
# Output: {'name': 'Alice', 'age': 31, 'profession': 'Engineer',
'salary': 70000, 'city': 'New York'}
```

### pop()

**Description:** Removes the specified key and returns its value. If the key is not found, it raises a `KeyError` unless a default value is provided.

```python
person = {"name": "Alice", "age": 30, "profession":
"Engineer"}

# Popping an existing key
profession = person.pop("profession")
print(profession)  # Output: Engineer
print(person)      # Output: {'name': 'Alice', 'age': 30}

# Popping a non-existent key without default (Raises
KeyError)
```

```
# salary = person.pop("salary")  # Raises KeyError: 'salary'

# Popping a non-existent key with default
salary = person.pop("salary", 50000)
print(salary)  # Output: 50000
print(person)  # Output: {'name': 'Alice', 'age': 30}
```

**popitem()**

**Description:** Removes and returns an arbitrary key-value pair as a tuple. As of Python 3.7, it removes the last inserted key-value pair.

```
person = {"name": "Alice", "age": 30, "profession":
"Engineer"}

# Popping the last item
item = person.popitem()
print(item)   # Output: ('profession', 'Engineer')
print(person) # Output: {'name': 'Alice', 'age': 30}
```

**clear()**

**Description:** Removes all items from the dictionary, resulting in an empty dictionary.

```
person = {"name": "Alice", "age": 30, "profession":
"Engineer"}
```

```
person.clear()
print(person)  # Output: {}
```

**setdefault()**
**Description:** Returns the value of a key if it exists; otherwise, inserts the key with a specified default value and returns that value.

# *Advanced Dictionary Concepts*

## Nested Dictionaries

**Description:** Dictionaries can contain other dictionaries as values, allowing for complex, multi-level data structures.

```
# Nested dictionary
company = {
    "employees": {
        "Alice": {"age": 30, "profession": "Engineer"},
        "Bob": {"age": 25, "profession": "Designer"},
        "Charlie": {"age": 35, "profession": "Manager"}
    },
    "departments": ["Engineering", "Design", "Management"]
}
```

# Accessing nested values
print(company["employees"]["Alice"]["profession"])  # Output: Engineer

# Adding a new employee
company["employees"]["Diana"] = {"age": 28, "profession": "Analyst"}
print(company["employees"])
# Output:
# {
#     'Alice': {'age': 30, 'profession': 'Engineer'},
#     'Bob': {'age': 25, 'profession': 'Designer'},
#     'Charlie': {'age': 35, 'profession': 'Manager'},
#     'Diana': {'age': 28, 'profession': 'Analyst'}
# }

**Dictionary Comprehensions**
**Description:** Similar to list and set comprehensions, dictionary comprehensions provide a concise way to create dictionaries based on existing iterables.

**Syntax:**
{key_expression: value_expression for item in iterable if condition}

**Examples:**

**Creating a Dictionary of Squares:**

```
squares = {x: x**2 for x in range(6)}
print(squares)
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

**Filtering Items:**

```
numbers = range(10)
even_squares = {x: x**2 for x in numbers if x % 2 == 0}
print(even_squares)
# Output: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

**Swapping Keys and Values:**

```
original = {"a": 1, "b": 2, "c": 3}
swapped = {value: key for key, value in original.items()}
print(swapped)
# Output: {1: 'a', 2: 'b', 3: 'c'}
```

**Note:** If the original dictionary has duplicate values, the resulting dictionary will only contain one of the possible key-value pairs, as keys must be unique.

## Default Dictionaries

**Description:** The `defaultdict` class from the `collections` module provides dictionaries with default values for non-

existent keys, avoiding the need to check for key existence.
**Advantages:**
• Automatically initializes missing keys with a default value.
• Simplifies code by eliminating the need for key existence checks.

**Common Use Cases:**
• **Counting:** Tallying occurrences of items.
```
from collections import defaultdict

words = ["apple", "banana", "apple", "cherry", "banana",
"apple"]
word_counts = defaultdict(int)

for word in words:
    word_counts[word] += 1

print(word_counts)
# Output: defaultdict(<class 'int'>, {'apple': 3, 'banana': 2,
'cherry': 1})
```

**Grouping: Grouping items based on a key.**
```
from collections import defaultdict

students = [
    {"name": "Alice", "grade": "A"},
```

```
    {"name": "Bob", "grade": "B"},
    {"name": "Charlie", "grade": "A"},
    {"name": "Diana", "grade": "C"},
    {"name": "Eve", "grade": "B"}
]

grade_groups = defaultdict(list)

for student in students:
    grade_groups[student["grade"]].append(student["name"])

print(grade_groups)
# Output:
# defaultdict(<class 'list'>, {'A': ['Alice', 'Charlie'], 'B': ['Bob',
'Eve'], 'C': ['Diana']})
```

**Important Notes:**

- The `default_factory` function defines the default value for non-existent keys. It should be callable without arguments.

- Once a `defaultdict` is created, you can modify its `default_factory` attribute if needed.

## *Examples and Exercises*

**Example 1: Storing and Retrieving User Information**

```
# Creating a dictionary to store user information
user = {
    "username": "alice123",
    "email": "alice@example.com",
    "age": 30,
    "is_active": True
}

# Accessing values
print(user["username"])  # Output: alice123
print(user.get("email")) # Output: alice@example.com

# Adding a new key-value pair
user["profession"] = "Engineer"

# Updating an existing key
user["age"] = 31

# Removing a key-value pair
```

```python
user.pop("is_active")

print(user)
# Output: {'username': 'alice123', 'email':
'alice@example.com', 'age': 31, 'profession': 'Engineer'}
```

**Example 2: Counting Occurrences Using Dictionaries**

```python
words = ["apple", "banana", "apple", "cherry", "banana",
"apple"]
word_counts = {}

for word in words:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1

print(word_counts)
# Output: {'apple': 3, 'banana': 2, 'cherry': 1}

# Using defaultdict for simplification
from collections import defaultdict

word_counts = defaultdict(int)
for word in words:
    word_counts[word] += 1
```

```python
print(word_counts)
# Output: defaultdict(<class 'int'>, {'apple': 3, 'banana': 2, 'cherry': 1})
```

## Example 3: Merging Dictionaries

```python
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

# Using the `update()` method
merged = dict1.copy()
merged.update(dict2)
print(merged)
# Output: {'a': 1, 'b': 3, 'c': 4}

# Using dictionary unpacking (Python 3.5+)
merged = {**dict1, **dict2}
print(merged)
# Output: {'a': 1, 'b': 3, 'c': 4}
```

### Example 4: Nested Dictionaries

```python
# Nested dictionary representing students and their grades
students = {
    "Alice": {"math": 90, "science": 85},
    "Bob": {"math": 75, "science": 95},
```

```
    "Charlie": {"math": 85, "science": 80}
}

# Accessing nested values
print(students["Alice"]["math"])  # Output: 90

# Adding a new subject for Bob
students["Bob"]["english"] = 88

print(students)
# Output:
# {
#     'Alice': {'math': 90, 'science': 85},
#     'Bob': {'math': 75, 'science': 95, 'english': 88},
#     'Charlie': {'math': 85, 'science': 80}
# }
```

## Exercise 1: Creating and Accessing a Dictionary

**Task:** Create a dictionary to store information about a book, including its title, author, publication year, and genres. Then, access and print the author's name.

**Solution:**

```
# Creating the dictionary
book = {
```

```python
    "title": "To Kill a Mockingbird",
    "author": "Harper Lee",
    "publication_year": 1960,
    "genres": ["Fiction", "Classic", "Coming-of-age"]
}

# Accessing the author's name
author = book["author"]
print(author)  # Output: Harper Lee
```

## Exercise 2: Updating and Removing Dictionary Entries

**Task:** Given the following dictionary representing a movie, update its rating to 8.5 and remove the director from the dictionary.

```python
movie = {
    "title": "Inception",
    "director": "Christopher Nolan",
    "year": 2010,
    "rating": 8.0
}
```

**Expected Output:**

```
{
    "title": "Inception",
    "year": 2010,
    "rating": 8.5
}
```

**Solution:**

```
movie = {
    "title": "Inception",
    "director": "Christopher Nolan",
    "year": 2010,
    "rating": 8.0
}

# Updating the rating
movie["rating"] = 8.5

# Removing the director
movie.pop("director")

print(movie)
# Output: {'title': 'Inception', 'year': 2010, 'rating': 8.5}
```

## Exercise 3: Using `.get()` with Default Values

**Task:** Given a dictionary representing a car, retrieve the value of the key "color". If the key does not exist, return "Unknown".

```
car = {
    "make": "Toyota",
    "model": "Camry",
    "year": 2020
}
```

**Expected Output:**
Unknown

```
car = {
    "make": "Toyota",
    "model": "Camry",
    "year": 2020
}

color = car.get("color", "Unknown")
print(color)  # Output: Unknown
```

## Exercise 4: Iterating Through a Dictionary

**Task:** Given a dictionary of students and their scores, print each student's name along with their score.

```
students_scores = {
    "Alice": 85,
    "Bob": 92,
    "Charlie": 78,
    "Diana": 90
}
```

**Expected Output:**

```
Alice: 85
Bob: 92
Charlie: 78
Diana: 90
```

**Solution:**

```
students_scores = {
    "Alice": 85,
    "Bob": 92,
    "Charlie": 78,
    "Diana": 90
}

for student, score in students_scores.items():
    print(f"{student}: {score}")
```

## Exercise 5: Merging Multiple Dictionaries

**Task:** Given a list of dictionaries representing various products, merge them into a single dictionary. If there are duplicate keys, later entries should overwrite earlier ones.

**Solution:**

Assuming that the `"id"` is the unique key.

```
products = [
    {"id": 1, "name": "Laptop", "price": 999},
    {"id": 2, "name": "Smartphone", "price": 499},
    {"id": 1, "name": "Gaming Laptop", "price": 1299}
]

merged_products = {}
for product in products:
    merged_products[product["id"]] = product

print(merged_products)
# Output:
# {
#     1: {'id': 1, 'name': 'Gaming Laptop', 'price': 1299},
#     2: {'id': 2, 'name': 'Smartphone', 'price': 499}
# }
```

# *Dictionary Unpacking Operator*

**Understanding the `**` Operator in Dictionary Merging**
In Python, the `**` operator is known as the **dictionary unpacking** operator. When used within a dictionary literal (i.e., inside `{}`), it **unpacks** the key-value pairs from one dictionary into another. This feature was introduced in **Python 3.5** and has become a popular and concise way to merge dictionaries.

## Syntax Breakdown

merged = {**dict1, **dict2}

- `{}`: Denotes a dictionary literal.
- `**dict1`: Unpacks all key-value pairs from `dict1` into the new dictionary.
- `**dict2`: Unpacks all key-value pairs from `dict2` into the new dictionary.
- `merged`: The resulting dictionary containing combined key-value pairs from both `dict1` and `dict2`.

## How It Works

1. **Unpacking Key-Value Pairs**: The `**` operator takes all the key-value pairs from `dict1` and `dict2` and inserts them into the new dictionary.
2. **Handling Duplicate Keys**: If both dictionaries contain the same key, the value from the latter dictionary (`dict2` in this case) **overwrites** the value from the former (`dict1`).

## How It Works

1. **Unpacking Key-Value Pairs**: The `**` operator takes all the key-value pairs from `dict1` and `dict2` and inserts them into the new dictionary.
2. **Handling Duplicate Keys**: If both dictionaries contain the same key, the value from the latter dictionary (`dict2` in this case) **overwrites** the value from the former (`dict1`).

```
# Define two dictionaries
dict1 = {
    "a": 1,
    "b": 2,
    "c": 3
}

dict2 = {
    "b": 20,
    "d": 4
```

}

# Merge dictionaries using ** unpacking
merged = {**dict1, **dict2}

print(merged)
# Output: {'a': 1, 'b': 20, 'c': 3, 'd': 4}

**Explanation:**
• The key `"b"` exists in both `dict1` and `dict2`.
• In the `merged` dictionary, the value of `"b"` is `20`, which comes from `dict2`, effectively **overwriting** the value `2` from `dict1`

## Advantages of Using `**` for Merging

1. **Conciseness**: The syntax is short and easy to read.
2. **Immutability**: It creates a **new dictionary** without modifying the original dictionaries, preserving data integrity.
3. **Flexibility**: You can merge more than two dictionaries seamlessly.

## Merging Multiple Dictionaries

You can merge more than two dictionaries by chaining the `**` operator:

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 20, "c": 3}
dict3 = {"c": 30, "d": 4}

merged = {**dict1, **dict2, **dict3}

print(merged)
# Output: {'a': 1, 'b': 20, 'c': 30, 'd': 4}
```

**Comparison with Other Merging Methods**

## Using the `.update()` Method

Another common way to merge dictionaries is using the `.update()` method. However, there are key differences:

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 20, "c": 3}

# Using .update() modifies dict1 in-place
dict1.update(dict2)
print(dict1)
# Output: {'a': 1, 'b': 20, 'c': 3}
```

**Key Differences:**

| Aspect | `**` Unpacking | `.update()` Method |
|---|---|---|
| **Creates New Dict** | Yes, does not modify original dicts | No, modifies the dictionary in-place |
| **Immutability** | Preserves original dictionaries | Alters the first dictionary directly |
| **Return Value** | Returns the new merged dictionary | Returns `None` |
| **Readability** | More concise and readable for merging | More verbose and intended for updating existing dicts |

**Example:**

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 20, "c": 3}

# Using ** unpacking
merged = {**dict1, **dict2}
print(merged)
# Output: {'a': 1, 'b': 20, 'c': 3}

print(dict1)
# Output: {'a': 1, 'b': 2}  # Unchanged

# Using .update()
dict1.update(dict2)
print(dict1)
# Output: {'a': 1, 'b': 20, 'c': 3}  # Modified
```

## Important Considerations

1. **Order Matters**: In cases where multiple dictionaries have the same keys, the **last dictionary's value** for that key will prevail.

```
dict1 = {"x": 1}
dict2 = {"x": 2}
dict3 = {"x": 3}

merged = {**dict1, **dict2, **dict3}
print(merged)
# Output: {'x': 3}
```

**Python Version**: The `**` unpacking for dictionaries within dictionary literals is available from **Python 3.5** onwards. Ensure your Python environment is compatible.

**Non-Dictionary Iterables**: While `.update()` can accept any iterable of key-value pairs, `**` unpacking specifically requires dictionaries or objects that implement the mapping protocol.

```
# Using .update() with an iterable of tuples
dict1 = {"a": 1}
dict1.update([("b", 2), ("c", 3)])
print(dict1)
```

# Output: {'a': 1, 'b': 2, 'c': 3}

# Using ** unpacking with non-dictionary iterables raises TypeError
try:
    merged = {**dict1, **[("d", 4)]}
except TypeError as e:
    print(e)  # Output: 'mapping expected instead of list'

# Additional Examples

## Example 1: Merging Dictionaries with Overlapping Keys

```
dict1 = {"name": "Alice", "age": 25, "city": "New York"}
dict2 = {"age": 30, "profession": "Engineer"}

merged = {**dict1, **dict2}
print(merged)
# Output: {'name': 'Alice', 'age': 30, 'city': 'New York', 'profession': 'Engineer'}
```

Example 2: Merging Dictionaries with No Overlapping Keys

```
dict1 = {"fruit": "apple"}
dict2 = {"vegetable": "carrot"}

merged = {**dict1, **dict2}
print(merged)
# Output: {'fruit': 'apple', 'vegetable': 'carrot'}
```

**Example 3: Merging Dictionaries in a Function**

```
def merge_dicts(*dicts):
    merged = {}
    for d in dicts:
        merged = {**merged, **d}
    return merged

dict_a = {"a": 1, "b": 2}
dict_b = {"b": 3, "c": 4}
dict_c = {"c": 5, "d": 6}

result = merge_dicts(dict_a, dict_b, dict_c)
print(result)
# Output: {'a': 1, 'b': 3, 'c': 5, 'd': 6}
```

**Example 4: Merging Dictionaries with Variable Keys**

```
defaults = {
    "host": "localhost",
    "port": 8080,
    "debug": False
```

```python
}

overrides = {
    "port": 9090,
    "debug": True
}

config = {**defaults, **overrides}
print(config)
# Output: {'host': 'localhost', 'port': 9090, 'debug': True}
```

## Comparing with Other Unpacking Scenarios

```python
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

info = {"name": "Bob", "age": 25}

greet(**info)
# Output: Hello, Bob! You are 25 years old.
```

**Explanation:**

• The `**info` syntax unpacks the `info` dictionary into keyword arguments, effectively calling `greet(name="Bob", age=25)`.

### Key Difference

◇ **In Function Calls**: `**` unpacks a dictionary into keyword arguments.
◇ **In Dictionary Literals**: `**` unpacks key-value pairs into a new dictionary.

# *Functions*

## Introduction to Functions

A **function** in Python is a block of reusable code that performs a specific task. Functions help in breaking down complex problems into smaller, manageable pieces, promoting code reusability and readability.

**Key Benefits of Using Functions:**

- **Reusability:** Write code once and reuse it multiple times.
- **Organization:** Break down complex tasks into simpler, manageable parts.
- **Readability:** Improve code clarity by encapsulating functionality.

• **Maintainability:** Easier to update and manage code by modifying functions.

**Terminology:**
◇ **Function Definition:** The code that specifies what the function does.
◇ **Function Call:** Executing the function to perform its task.
◇ **Parameters (Arguments):** Inputs passed to a function to customize its behavior.
◇ **Return Value:** The output produced by a function.

# *Function Basics*

## Defining a Function

In Python, functions are defined using the `def` keyword, followed by the function name and parentheses `()`. The function body is indented and contains the code to be executed.

**Syntax:**

```
def function_name(parameters):
    """Docstring explaining the function."""
    # Function body
```

```
    return value
```

**Example:**

```
def greet():
    """Prints a greeting message."""
    print("Hello, World!")
```

## Calling a Function

Once a function is defined, you can execute it by **calling** its name followed by parentheses. If the function requires parameters, you pass them inside the parentheses.

**Syntax:**

```
function_name(arguments)
```

**Example:**

```
greet()  # Output: Hello, World!
```

## Function Parameters

Parameters allow you to pass data to functions, making them more flexible and dynamic. Python supports various types of parameters:

#### Positional Arguments

Arguments passed to a function based on their position.

```
def add(a, b):
    return a + b

result = add(5, 3)
print(result)  # Output: 8
```

### Keyword Arguments

Arguments passed to a function by explicitly specifying the parameter name.

```
def describe_person(name, age):
    print(f"{name} is {age} years old.")

describe_person(age=25, name="Alice")  # Output: Alice is 25 years old.
```

### Default Arguments

Parameters that assume a default value if no argument is provided.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()         # Output: Hello, Guest!
greet("Bob")    # Output: Hello, Bob!
```

**Variable-Length Arguments**

Allows a function to accept an arbitrary number of arguments.

- `*args`: Non-keyword variable-length arguments.
- `**kwargs`: Keyword variable-length arguments.

**Example:**

## Non-keyword (Positional) Variable-length Arguments

- `*args` allows you to pass **any number of positional arguments** (i.e., arguments without a keyword) to a function. It collects these arguments into a **tuple**.
- You use `*args` when you does not know how many arguments are going to be passed to the function in advance

**Example:**

```
def multiply(*args):
    result = 1
    for number in args:
        result *= number
    return result

print(multiply(2, 3))        # Output: 6
print(multiply(2, 3, 4))     # Output: 24
```

## `**kwargs`: Keyword Variable-length Arguments

- `**kwargs` allows you to pass **any number of keyword arguments** (i.e., arguments with a key and a value, like `key=value`) to a function. It collects these arguments into a **dictionary**.
- You use `**kwargs` when you want to pass a variable number of keyword arguments.

**Example:**

```
def build_profile(first, last, **kwargs):
    profile = {}
    profile['first_name'] = first
    profile['last_name'] = last
    for key, value in kwargs.items():
        profile[key] = value
```

```
    return profile

user_profile = build_profile('John', 'Doe', location='USA',
field='Engineering')
print(user_profile)
# Output: {'first_name': 'John', 'last_name': 'Doe', 'location':
'USA', 'field': 'Engineering'}
```

## Return Values

Functions can return values using the `return` statement. If no `return` statement is provided, the function returns `None` by default.

**Example:**

```
def square(x):
    return x * x

result = square(4)
print(result)  # Output: 16
```

**Returning Multiple Values:**

Python functions can return multiple values as a tuple.

```
def get_coordinates():
    x = 10
```

```
    y = 20
    return x, y

coords = get_coordinates()
print(coords)       # Output: (10, 20)
print(type(coords))  # Output: <class 'tuple'>
```

# *Advanced Function Concepts*

## Anonymous Functions (`lambda`)

**Lambda functions** are small, unnamed functions defined using the `lambda` keyword. They are primarily used for short, throwaway functions.

**Syntax:**

```
lambda arguments: expression
```

**Example:**

```
add = lambda a, b: a + b
print(add(5, 3))  # Output: 8
```

**Use Cases:**

◇ **Sorting with Custom Keys:**

```
students = [("Alice", 25), ("Bob", 20), ("Charlie", 23)]
# Sort by age
sorted_students = sorted(students, key=lambda student:
student[1])
print(sorted_students)
# Output: [('Bob', 20), ('Charlie', 23), ('Alice', 25)]
```

# Use Cases:

**Sorting with Custom Keys:**


```
students = [("Alice", 25), ("Bob", 20), ("Charlie", 23)]
# Sort by age
sorted_students = sorted(students, key=lambda student:
student[1])
print(sorted_students)
# Output: [('Bob', 20), ('Charlie', 23), ('Alice', 25)]
```


**Using with `map()`, `filter()`, and `reduce()`**

**Limitations:**

• **Single Expression:** Lambda functions can only contain a single expression, making them unsuitable for complex

operations.
• **No Statements:** They cannot contain statements like loops or multiple expressions.

## Higher-Order Functions

**Higher-order functions** are functions that take other functions as arguments or return them as results. They enable functional programming paradigms in Python.

**Common Higher-Order Functions:**
• `map()`: Applies a function to all items in an iterable.
• `filter()`: Filters items out of an iterable based on a function.
• `reduce()`: Applies a function cumulatively to the items of an iterable (requires `functools`).
• `sorted()`: Sorts items using a key function.

## Decorators

**Decorators** are a powerful feature in Python that allow you to modify or enhance functions without changing their actual code. They are higher-order functions that take a function as an argument and return a new function.

**Syntax:**

```
@decorator_name
def function_to_decorate():
    pass
```

**Basic Example:**

```
def my_decorator(func):
    def wrapper():
        print("Before the function runs.")
        func()
        print("After the function runs.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Before the function runs.
# Hello!
# After the function runs.
```

**Use Cases:**

- **Logging**
- **Access Control**
- **Caching**

• **Timing Functions**

**Example: Logging Decorator:**

```
def my_decorator(func):
    def wrapper():
        print("Before the function runs.")
        func()
        print("After the function runs.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Before the function runs.
# Hello!
# After the function runs.
```

## Recursion

**Recursion** is a programming technique where a function calls itself to solve smaller instances of a problem. It is particularly useful for tasks that can be divided into similar subtasks.

**Key Components of Recursion:**
1. **Base Case:** The condition under which the recursion stops.
2. **Recursive Case:** The part of the function where it calls itself with modified parameters.

**Example: Factorial Function**

```
def factorial(n):
    """Returns the factorial of n."""
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n - 1)  # Recursive case

print(factorial(5))  # Output: 120

"""
How it work:

Note:
{} => indicate return value
[] => function call



[factorial(5)]
{5 * [factorial(4)]}
5 * {4* [factorial(3)]}
```

20 * {3 * [factorial(2)]}
60 * {2 * [factorial(1)]}
120 * {1 * [factorial(0)]}
120 * 1
"""

**Example: Fibonacci Sequence**

```
def fibonacci(n):
    """Returns the nth Fibonacci number."""
    if n <= 0:
        return 0  # Base case
    elif n == 1:
        return 1  # Base case
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive case

print(fibonacci(6))  # Output: 8
```

"""
How It work:

Note:
{} => indicate return value
[] => function call

fibonacci(6)
{[fibonacci(5)] + fibonacci(4)}
{[fibonacci(4)] + fibonacci(3)} + fibonacci(4)
{[fibonacci(3)] + fibonacci(2)} + fibonacci(3) + fibonacci(4)
{[fibonacci(2)] + fibonacci(1)} fibonacci(2) + fibonacci(3) + fibonacci(4)
{[fibonacci(1)] + fibonacci(0)} + fibonacci(1) + fibonacci(2) + fibonacci(3) + fibonacci(4)

{1} + [fibonacci(0)}] + fibonacci(1) + fibonacci(2) + fibonacci(3) + fibonacci(4)
1 + {0} + [fibonacci(1)]+ fibonacci(2) + fibonacci(3) + fibonacci(4)
1 + {1} + [fibonacci(2)] + fibonacci(3) + fibonacci(4)
2 + {[fibonacci(1)] + fibonacci(0)} + fibonacci(3) + fibonacci(4)
2 + {1} + [fibonacci(0)] + fibonacci(3) + fibonacci(4)
3 + {0} + [fibonacci(3)] + fibonacci(4)
3 + {[fibonacci(2)] + fibonacci(1)} + fibonacci(4)
3 + {[fibonacci(1)] + fibonacci(0)} + fibonacci(1)} + fibonacci(4)
3 + {1} + [fibonacci(0)] + fibonacci(1)} + fibonacci(4)
4 + {0} + [fibonacci(1)] + fibonacci(4)
4 + {1} + [fibonacci(4)]
5 + {[fibonacci(3)] + fibonacci(2)}
5 + {[fibonacci(2)] + fibonacci(1)} + fibonacci(2)

```
5 + {[fibonacci(1)] + fibonacci(0)} + fibonacci(1) + fibonacci(2)
5 + {1} +  [fibonacci(0)] + fibonacci(1) + fibonacci(2)
6 + {0} + [fibonacci(1)] + fibonacci(2)
6 + {1} + [fibonacci(2)]
7 + {[fibonacci(1)] + fibonacci(0)}
7 + {1} + [fibonacci(0)]
8 + [fibonacci(0)]
8 + {0}
8
"""
```

**Considerations:**

- **Stack Overflow:** Deep recursion can lead to stack overflow errors. Python has a recursion limit (`sys.getrecursionlimit()`) that can be adjusted with caution.
- **Efficiency:** Recursive solutions may be less efficient for certain problems due to repeated computations. Techniques like memoization can help optimize recursive functions.

## Generator Functions

**Generators** are a special type of function that return an iterator, allowing you to iterate over a sequence of values lazily (i.e., one at a time and only when needed). They are memory-efficient and suitable for handling large datasets.

**Defining a Generator:**
Use the `yield` keyword instead of `return` to produce a value.

**Example: Simple Generator**

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for number in countdown(5):
    print(number)
# Output:
# 5
# 4
# 3
# 2
# 1
```

## Closures

A **closure** occurs when a nested function captures the local variables from its enclosing scope. Closures enable the creation of functions with preserved state.

**Example:**

```
def make_multiplier(factor):
    def multiplier(x):
        return x * factor
    return multiplier

times_three = make_multiplier(3)
print(times_three(5))  # Output: 15

times_five = make_multiplier(5)
print(times_five(5))   # Output: 25
```

**Explanation:**

- `make_multiplier` is a higher-order function that returns the `multiplier` function.
- The `multiplier` function retains access to the `factor` variable from its enclosing scope, even after `make_multiplier` has finished executing.

# *Function Scope and Lifetime*

Understanding variable scope within functions is crucial to prevent unexpected behaviors and bugs.

## Local and Global Variables

- **Local Variables:** Defined within a function and accessible only inside that function.
- **Global Variables:** Defined outside of any function and accessible throughout the module.

Example:

```
x = 10  # Global variable

def foo():
    y = 5  # Local variable
    print(f"Inside foo: x = {x}, y = {y}")

foo()
print(f"Outside foo: x = {x}")
# Output:
# Inside foo: x = 10, y = 5
# Outside foo: x = 10
```

**Note:** Attempting to access a local variable outside its function raises a `NameError`.

```
def bar():
    z = 20

bar()
print(z)  # Raises NameError: name 'z' is not defined
```

# *Examples and Exercises*

## Example 1: Basic Function

**Task:** Define a function that prints a welcome message.

```
def welcome():
    """Prints a welcome message."""
    print("Welcome to Python Functions!")

welcome()  # Output: Welcome to Python Functions!
```

## Example 2: Function with Multiple Parameters

**Task:** Define a function that takes two numbers and returns their sum.

```
def add(a, b):
    """Returns the sum of a and b."""
    return a + b

result = add(5, 7)
print(result)  # Output: 12
```

## Example 3: Using Default Parameters

**Task:** Define a function that greets a user, with a default name of "Guest".

```
def greet(name="Guest"):
    """Greets the user by name."""
    print(f"Hello, {name}!")

greet()          # Output: Hello, Guest!
greet("Alice")   # Output: Hello, Alice!
```

## Example 4: Recursive Function

**Task:** Define a recursive function to compute the factorial of a number.

```
def factorial(n):
    """Returns the factorial of n."""
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n - 1)  # Recursive case
```

```python
print(factorial(5))  # Output: 120
```

### Exercise 1: Factorial Function

**Task:** Write a function `compute_factorial(n)` that returns the factorial of `n` using recursion.

```python
def compute_factorial(n):
    """Computes the factorial of n recursively."""
    if n < 0:
        return "Factorial not defined for negative numbers."
    elif n == 0 or n == 1:
        return 1
    else:
        return n * compute_factorial(n - 1)

# Testing the function
print(compute_factorial(5))  # Output: 120
print(compute_factorial(0))  # Output: 1
print(compute_factorial(-3)) # Output: Factorial not defined for negative numbers.
```

**Solution:**

```python
def compute_factorial(n):
    """Computes the factorial of n recursively."""
```

```
    if n < 0:
        return "Factorial not defined for negative numbers."
    elif n == 0 or n == 1:
        return 1
    else:
        return n * compute_factorial(n - 1)

# Testing the function
print(compute_factorial(5))  # Output: 120
print(compute_factorial(0))  # Output: 1
print(compute_factorial(-3)) # Output: Factorial not defined for negative numbers.
```

## Exercise 2: Fibonacci Sequence

**Task:** Write a function `fibonacci(n)` that returns the nth Fibonacci number using recursion.

**Solution**

```
def fibonacci(n):
    """Returns the nth Fibonacci number."""
    if n < 0:
        return "Fibonacci number not defined for negative indices."
    elif n == 0:
        return 0
    elif n == 1:
```

```
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Testing the function
print(fibonacci(6))  # Output: 8
print(fibonacci(0))  # Output: 0
print(fibonacci(-2)) # Output: Fibonacci number not defined
for negative indices.
```

## Exercise 3: Decorator for Logging

**Task:** Create a decorator `logger` that logs the function name and its arguments each time the function is called.

Solution:

```
def logger(func):
    """Decorator that logs the function call details."""
    def wrapper(*args, **kwargs):
        args_str = ', '.join([str(a) for a in args])
        kwargs_str = ', '.join([f"{k}={v}" for k, v in
kwargs.items()])
        print(f"Calling '{func.__name__}' with args: {args_str}
and kwargs: {kwargs_str}")
        result = func(*args, **kwargs)
        print(f"'{func.__name__}' returned {result}")
```

```
        return result
    return wrapper

@logger
def multiply(a, b):
    return a * b

# Testing the decorator
multiply(3, 4)
# Output:
# Calling 'multiply' with args: 3, 4 and kwargs:
# 'multiply' returned 12
```

## Exercise 4: Generator for Even Numbers

**Task:** Write a generator function `even_numbers(limit)` that yields even numbers up to a specified limit.

**Solution:**

```
def even_numbers(limit):
    """Yields even numbers up to the specified limit."""
    for num in range(2, limit + 1, 2):
        yield num

# Using the generator
for even in even_numbers(10):
    print(even)
```

# Output:
# 2
# 4
# 6
# 8
# 10

## *time and datetime Module*

### Introduction to `time` and `datetime` Modules

Python offers two primary modules for handling dates and times:

• `time` **Module:**

◇ Provides functions for working with time-related tasks.

◇ Focuses on low-level time operations.

◇ Returns time in seconds since the epoch (January 1, 1970).

• `datetime` **Module:**

◇ Offers more advanced and user-friendly classes for handling dates and times.

◇ Supports date arithmetic, timezone handling, and more.

◇ Recommended for most date and time operations due to its versatility.

**Key Differences:**

| Feature | `time` Module | `datetime` Module |
|---|---|---|
| **Primary Classes** | `struct_time` | `datetime`, `date`, `time`, `timedelta`, `timezone` |
| **Time Representation** | Seconds since the epoch | Combination of date and time |
| **Use Cases** | Low-level time operations, performance-critical tasks | High-level date and time manipulations, formatting, arithmetic |

# *Understanding Time Tuples*

A **time tuple** is a named tuple that represents a specific moment in time. It's returned by several functions in the `t-ime` module.

## Components of `struct_time`

The `struct_time` tuple has the following attributes:

The `struct_time` tuple has the following attributes:

| Attribute | Description |
|---|---|
| `tm_year` | Year (e.g., 2024) |
| `tm_mon` | Month (1-12) |
| `tm_mday` | Day of the month (1-31) |
| `tm_hour` | Hour (0-23) |
| `tm_min` | Minute (0-59) |
| `tm_sec` | Second (0-60, allowing for leap seconds) |
| `tm_wday` | Day of the week (0-6, Monday is 0) |
| `tm_yday` | Day of the year (1-366) |
| `tm_isdst` | Daylight Saving Time flag (-1, 0, 1) |

**Creating and Accessing `struct_time`**

```
import time

# Get the current local time as a struct_time
current_time = time.localtime()
print(current_time)
# Output: time.struct_time(tm_year=2024, tm_mon=10,
tm_mday=8, tm_hour=12, tm_min=30, tm_sec=45,
tm_wday=0, tm_yday=282, tm_isdst=1)

# Access individual attributes
print("Year:", current_time.tm_year)    # Output: Year: 2024
print("Month:", current_time.tm_mon)    # Output: Month: 10
print("Day:", current_time.tm_mday)     # Output: Day: 8
```

Common `time` Module Functions Returning `struct_time`

• `time.localtime([secs])`: Converts seconds since the epoch to `struct_time` in local time.
• `time.gmtime([secs])`: Converts seconds since the epoch to `struct_time` in UTC.
• `time.strptime(string, format)`: Parses a string into `struct_time` based on the specified format.

**Example: Parsing a Date String**

```
import time

date_string = "2024-10-08 12:30:45"
format_string = "%Y-%m-%d %H:%M:%S"

parsed_time = time.strptime(date_string, format_string)
print(parsed_time)
# Output: time.struct_time(tm_year=2024, tm_mon=10,
tm_mday=8, tm_hour=12, tm_min=30, tm_sec=45,
tm_wday=0, tm_yday=282, tm_isdst=-1)
```

## *Date and Time Formatting (strftime and strptime)*

Formatting dates and times allows you to convert `struct_time` or `datetime` objects to strings and vice versa. Python provides two primary methods:

- `strftime`: Converts a `struct_time` or `datetime` object to a formatted string.
- `strptime`: Parses a string into a `struct_time` or `datetime` object based on a format.

**Common Format Codes**

| Code | Meaning | Example |
|---|---|---|
| `%Y` | Year with century | 2024 |
| `%y` | Year without century | 24 |
| `%m` | Month as a zero-padded decimal | 01 to 12 |
| `%d` | Day of the month as zero-padded decimal | 01 to 31 |
| `%H` | Hour (24-hour clock) as zero-padded decimal | 00 to 23 |
| `%I` | Hour (12-hour clock) as zero-padded decimal | 01 to 12 |
| `%p` | Locale's AM or PM | AM, PM |
| `%M` | Minute as a zero-padded decimal | 00 to 59 |
| `%S` | Second as a zero-padded decimal | 00 to 59 |
| `%A` | Full weekday name | Monday |
| `%a` | Abbreviated weekday name | Mon |
| `%B` | Full month name | January |
| `%b` | Abbreviated month name | Jan |
| `%%` | A literal `%` character | % |

## Using `strftime`

**Example: Formatting Current Local Time**

import time

current_time = time.localtime()

# Format: Year-Month-Day Hour:Minute:Second
formatted_time = time.strftime("%Y-%m-%d %H:%M:%S", current_time)
print(formatted_time)  # Output: 2024-10-08 12:30:45

Example: Full Date with Weekday and Month Name
formatted_full_date = time.strftime("%A, %B %d, %Y", current_time)
print(formatted_full_date)  # Output: Monday, October 08, 2024

## Using `strptime`

**Example: Parsing a Date String to** `struct_time`

import time

date_string = "Monday, October 08, 2024"
format_string = "%A, %B %d, %Y"

parsed_time = time.strptime(date_string, format_string)
print(parsed_time)
# Output: time.struct_time(tm_year=2024, tm_mon=10, tm_mday=8, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=0, tm_yday=282, tm_isdst=-1)

## *Building datetime Objects from Formatted Strings*

While the `time` module offers basic functionality, the `datetime` module provides a more intuitive and feature-rich way to handle dates and times.

**Importing `datetime`**

from datetime import datetime

### Parsing Strings to `datetime` Objects with `strptime`

**Example:**

from datetime import datetime

```
date_string = "2024-10-08 12:30:45"
format_string = "%Y-%m-%d %H:%M:%S"

# Parse string to datetime object
parsed_datetime = datetime.strptime(date_string,
format_string)
print(parsed_datetime)
# Output: 2024-10-08 12:30:45
```

## Formatting `datetime` Objects with `strftime`

**Example:**

```
from datetime import datetime

# Current datetime
now = datetime.now()

# Format: Day/Month/Year Hour:Minute AM/PM
formatted_now = now.strftime("%d/%m/%Y %I:%M %p")
print(formatted_now)  # Output: 08/10/2024 12:30 PM
```

## Building `datetime` Objects from Components

You can create `datetime` objects by specifying individual components like year, month, day, etc.

**Example:**

```
from datetime import datetime

# Create a datetime object for October 8, 2024, at 12:30:45 PM
custom_datetime = datetime(2024, 10, 8, 12, 30, 45)
print(custom_datetime)
# Output: 2024-10-08 12:30:45
```

## Handling Time Zones with `datetime`

Python's `datetime` module also supports timezone-aware objects using the `timezone` class or third-party libraries like `pytz`.

**Example Using `timezone`:**

```
from datetime import datetime, timezone, timedelta

# Current UTC time
utc_now = datetime.now(timezone.utc)
print(utc_now)
# Output: 2024-10-08 12:30:45.123456+00:00

# Create a timezone for UTC+5
tz_plus_5 = timezone(timedelta(hours=5))

# Current time in UTC+5
```

local_time = utc_now.astimezone(tz_plus_5)
print(local_time)
# Output: 2024-10-08 17:30:45.123456+05:00

Example Using `pytz`:
from datetime import datetime
import pytz

# Define timezone
eastern = pytz.timezone('US/Eastern')

# Current time in Eastern timezone
eastern_time = datetime.now(eastern)
print(eastern_time)
# Output: 2024-10-08 08:30:45.123456-04:00

*Note: To use `pytz`, you need to install it using `pip install pytz`.*

# ***Practical Examples***

## Example 1: Getting Current Date and Time

**Using `time` Module:**

```
import time

# Current time in seconds since epoch
current_epoch = time.time()
print(current_epoch)  # Output: e.g., 1702219845.123456

# Current local time as struct_time
local_time = time.localtime()
print(local_time)
# Output: time.struct_time(tm_year=2024, tm_mon=10,
tm_mday=8, tm_hour=12, tm_min=30, tm_sec=45,
tm_wday=0, tm_yday=282, tm_isdst=1)
```

Using `datetime` Module:

```
from datetime import datetime

# Current datetime
now = datetime.now()
print(now)
# Output: 2024-10-08 12:30:45.123456
```

## Example 2: Formatting Dates

**Convert `datetime` to String:**

```
from datetime import datetime
```

```
now = datetime.now()

# Format: Month-Day-Year Hour:Minute:Second
formatted = now.strftime("%m-%d-%Y %H:%M:%S")
print(formatted)  # Output: 10-08-2024 12:30:45
```

**Convert `struct_time` to String:**

```
import time

local_time = time.localtime()

# Format: Weekday, Month Day, Year
formatted = time.strftime("%A, %B %d, %Y", local_time)
print(formatted)  # Output: Monday, October 08, 2024
```

## Example 3: Parsing Strings to Dates

**Parse String to `datetime` Object:**

```
from datetime import datetime

date_string = "2024-10-08 12:30:45"
format_string = "%Y-%m-%d %H:%M:%S"

parsed_date = datetime.strptime(date_string, format_string)
```

```
print(parsed_date)
# Output: 2024-10-08 12:30:45
```

**Parse String to `struct_time`:**

```
import time

date_string = "Monday, October 08, 2024"
format_string = "%A, %B %d, %Y"

parsed_time = time.strptime(date_string, format_string)
print(parsed_time)
# Output: time.struct_time(tm_year=2024, tm_mon=10,
tm_mday=8, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=0,
tm_yday=282, tm_isdst=-1)
```

## Example 4: Calculating Time Differences

**Using `datetime`:**

```
from datetime import datetime

# Define two dates
date1 = datetime(2024, 10, 8, 12, 30, 45)
date2 = datetime(2024, 10, 10, 15, 45, 30)

# Calculate difference
```

difference = date2 - date1
print(difference)
# Output: 2 days, 3:14:45

**Using `timedelta`:**

```
from datetime import timedelta

# Create a timedelta of 5 days, 4 hours, and 30 minutes
delta = timedelta(days=5, hours=4, minutes=30)
print(delta)
# Output: 5 days, 4:30:00
```

### Example 5: Scheduling a Task

**Using `time.sleep` to Delay Execution:**

```
import time

print("Task started.")
time.sleep(5)  # Pause execution for 5 seconds
print("Task completed after 5 seconds.")
```

# ***Exercise***

To reinforce your understanding, here are some exercises.

Try solving them on your own before checking the solutions!

## Exercise 1: Current Time in Different Formats

**Task:** Write a Python script that prints the current date and time in the following formats:

1. `YYYY-MM-DD HH:MM:SS`
2. `Day, Month DD, YYYY`
3. `MM/DD/YYYY`
4. `HH:MM AM/PM`

Solution:

```
from datetime import datetime

now = datetime.now()

# Format 1: YYYY-MM-DD HH:MM:SS
format1 = now.strftime("%Y-%m-%d %H:%M:%S")
print("Format 1:", format1)

# Format 2: Day, Month DD, YYYY
format2 = now.strftime("%A, %B %d, %Y")
print("Format 2:", format2)

# Format 3: MM/DD/YYYY
format3 = now.strftime("%m/%d/%Y")
print("Format 3:", format3)
```

```
# Format 4: HH:MM AM/PM
format4 = now.strftime("%I:%M %p")
print("Format 4:", format4)
```

Output (Example):
Format 1: 2024-10-08 12:30:45
Format 2: Monday, October 08, 2024
Format 3: 10/08/2024
Format 4: 12:30 PM

**Task:** Write a Python program that calculates the number of days remaining until January 1st of the next year.

Solution:

```
from datetime import datetime

def days_until_new_year():
    today = datetime.now()
    current_year = today.year
    new_year = datetime(current_year + 1, 1, 1)
    delta = new_year - today
    return delta.days

print(f"Days until New Year: {days_until_new_year()}")
```

Output (Example):
Days until New Year: 84

## Exercise 3: Parse and Format Date from User Input

**Task:** Prompt the user to enter a date in the format `DD-MM-YYYY`, parse it into a `datetime` object, and then print it in the format `Month DD, YYYY`.

**Solution:**

```
from datetime import datetime

# Prompt user for input
date_input = input("Enter a date (DD-MM-YYYY): ")

# Define input and output formats
input_format = "%d-%m-%Y"
output_format = "%B %d, %Y"

try:
    # Parse input string to datetime object
    parsed_date = datetime.strptime(date_input,
input_format)

    # Format datetime object to desired string format
    formatted_date = parsed_date.strftime(output_format)
```

```
    print("Formatted Date:", formatted_date)
except ValueError:
    print("Invalid date format. Please use DD-MM-YYYY.")
```

**Sample Interaction:**
Enter a date (DD-MM-YYYY): 08-10-2024
Formatted Date: October 08, 2024

## Exercise 4: Timezone Conversion

**Task:** Write a Python script that gets the current UTC time and converts it to Eastern Standard Time (EST).

**Solution:**

```
from datetime import datetime, timezone, timedelta

# Current UTC time
utc_now = datetime.now(timezone.utc)
print("UTC Time:", utc_now.strftime("%Y-%m-%d %H:%M:%S %Z"))

# Define EST timezone (UTC-5)
est = timezone(timedelta(hours=-5))

# Convert UTC to EST
est_time = utc_now.astimezone(est)
print("EST Time:", est_time.strftime("%Y-%m-%d %H:%M:%S %Z"))
```

Output (Example):
UTC Time: 2024-10-08 17:30:45 UTC
EST Time: 2024-10-08 12:30:45 EST

## Exercise 5: Calculate Age from Birthdate

**Task:** Write a Python program that prompts the user to enter their birthdate in `YYYY-MM-DD` format and calculates their age in years.
**Solution:**

```
from datetime import datetime

def calculate_age(birthdate_str):
    # Define the input format
    format_str = "%Y-%m-%d"

    try:
        # Parse the birthdate string to a datetime object
        birthdate = datetime.strptime(birthdate_str, format_str)
        today = datetime.now()

        # Calculate age
        age = today.year - birthdate.year - ((today.month,
today.day) < (birthdate.month, birthdate.day))
        return age
```

```
    except ValueError:
        return "Invalid date format. Please use YYYY-MM-DD."

# Prompt user for input
birthdate_input = input("Enter your birthdate (YYYY-MM-DD): ")
age = calculate_age(birthdate_input)
print("Your Age:", age)
```

Sample Interaction:
Enter your birthdate (YYYY-MM-DD): 1990-05-15
Your Age: 34

Note: The actual age will vary based on the current date.

# *File Input/Output (I/O)*

File Input/Output (I/O) is a fundamental aspect of programming, enabling your Python applications to interact with the file system by reading from and writing to files. Whether you're processing data, logging events, or managing configurations, understanding File I/O in Python is essential.

# Introduction to File I/O in Python

**File I/O** refers to the process of reading from and writing to files. Python provides built-in functions and modules to handle file operations efficiently.

**Key Modules:**

• `os`: Interacts with the operating system, handling file and directory operations.
• `pathlib`: Offers an object-oriented approach to handle filesystem paths.
• `shutil`: Provides high-level file operations like copying and removing directories.

# *Listing Files in a Directory*

To interact with the file system, you often need to list files within directories. Python offers multiple ways to achieve this.

## Using the `os` Module

import os

```
# List all files and directories in the current directory
entries = os.listdir('.')
print(entries)
```

**Output:**
['file1.txt', 'file2.py', 'folder1', 'folder2']

**Filtering Only Files:**

```
import os

# List only files in the current directory
files = [f for f in os.listdir('.') if os.path.isfile(f)]
print(files)
```

**Output:**
['file1.txt', 'file2.py']

## Using the os.scandir() Method

```
 # It return posix.ScandirIterator object
file_dir = os.scandir(file_pah)

for file in file_dir:
    #file.name give the file name
    print(file.name)
```

```python
#separator
print()

# filter file from directory. This loop is not work because os.scandir() method return an iterable.
# Iterable behaviour is if You fully it there no more item exist

for file in file_dir:
    if file.is_file():
        print(file.name)




#  If you encounter the problem convert it to list
file_dir = list(os.scandir(file_pah))

for file in file_dir:
    print(file)



print()

# Print only file
file_list = [file.name for file in file_dir if file.is_file()]
print(file_list)
print()
```

```python
[print(file.name) for file in file_dir if file.is_file()]
```

## Using the `pathlib` Module

Introduced in Python 3.4, `pathlib` offers a more intuitive way to handle filesystem paths.

```python
from pathlib import Path

# Create a Path object for the current directory
current_dir = Path('.')

# List all files and directories
entries = list(current_dir.iterdir())
print(entries)
```

**Output:**

[PosixPath('file1.txt'), PosixPath('file2.py'), PosixPath('folder1'), PosixPath('folder2')]

**Filtering Only Files:**

```python
from pathlib import Path

current_dir = Path('.')
```

```
# List only files
files = [f for f in current_dir.iterdir() if f.is_file()]
print(files)
```

**Output:**

```
[PosixPath('file1.txt'), PosixPath('file2.py')]
```

# *File Access Modes ("r", "w", "a", etc.)*

When working with files, specifying the correct access mode is crucial. The mode determines how the file will be handledâ€”whether you're reading, writing, appending, etc.

| Mode | Description |
| --- | --- |
| `"r"` | **Read**: Opens a file for reading. The file must exist. |
| `"w"` | **Write**: Opens a file for writing. Creates the file if it doesn't exist or truncates it if it does. |
| `"a"` | **Append**: Opens a file for appending. Creates the file if it doesn't exist. |
| `"x"` | **Exclusive Creation**: Creates a new file and fails if the file already exists. |
| `"b"` | **Binary Mode**: Opens the file in binary mode (e.g., `"rb"`, `"wb"`). |
| `"t"` | **Text Mode**: Opens the file in text mode (default, e.g., `"rt"`, `"wt"`). |
| `"+"` | **Update**: Opens the file for updating (reading and writing, e.g., `"r+"`, `"w+"`, `"a+"`). |

**File Access mode:**

**Read Only ('r'):** This mode opens the text files for reading only.
It raises the I/O error if the file does not exist. This is the default mode for opening files as well.

**Read and Write ('r+')**: This method opens the file for both reading and writing.
If the file does not exist, an I/O error gets raised.

**Write Only ('w'):** This mode opens the file for writing only.
The data in existing files are modified and overwritten.
If the file does not already exist in the folder, a new one gets created.

**Write and Read ('w+'):** This mode opens the file for both reading and writing.
The text is overwritten and deleted from an existing file.

**Append Only ('a'):** This mode allows the file to be opened for writing.
If the file doesn't yet exist, a new one gets created.
The newly written data will be added at the end, following the previously written data.

**Append and Read ('a+'):** Using this method, you can read and write in the file.
If the file doesn't already exist, one gets created.
The newly written text will be added at the end, following the previously written data.

## *Opening Files*

To perform any file operation, you first need to open the file using the `open()` function.

## Syntax:

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

- **Parameters:**

◇ `file`: Path to the file.
◇ `mode`: Access mode (e.g., `"r"`, `"w"`).
◇ `encoding`: Specifies the encoding (e.g., `"utf-8"`). Important for text files.

- **Best Practice:** Use **context managers** (`with` statement) to ensure files are properly closed after operations.

## Using Context Managers:

```
# Using 'with' ensures the file is closed automatically
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

**Advantages of Context Managers:**

- Automatically handles file closing.
- Cleaner and more readable code.

• Prevents resource leaks.

## *Reading from Files*

Once a file is opened in read mode, you can extract its contents using various methods.

### Reading the Entire File:

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

### Reading Specific Number of Characters:

```
with open('example.txt', 'r') as file:
    content = file.read(10)  # Reads first 10 characters
    print(content)
```

Reading Lines into a List:

```
with open('example.txt', 'r') as file:
    lines = file.readlines()
    print(lines)
```

Output:

```
['First line.\n', 'Second line.\n', 'Third line.\n']
```

Iterating Over Each Line:

```
with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())  # .strip() removes newline characters
```

Output:[line in File]

```
First line.
Second line.
Third line.
```

# *Reading Files Line by Line*

Reading large files all at once can be memory-intensive. To handle such cases, read files **line by line**.

## Using a Loop:

```
with open('large_file.txt', 'r') as file:
    for line in file:
        process(line)  # Replace with your processing logic
```

Using `readline()`:

```
with open('large_file.txt', 'r') as file:
    while True:
        line = file.readline()
        if not line:
            break
        process(line)  # Replace with your processing logic
```

**Advantages:**
- Efficient memory usage.
- Suitable for processing streaming data or very large files.

# *Closing Files*

Properly closing files ensures that resources are freed and data is correctly written to disk.

## Manual Closing:

```
file = open('example.txt', 'r')
content = file.read()
print(content)
file.close()
```

## Automatic Closing with Context Managers:

As shown earlier, using `with` ensures automatic closing.

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
# File is automatically closed here
```

**Recommendation:** Always use context managers to handle files. It reduces the risk of forgetting to close files and manages exceptions gracefully.

## *Writing to Files*

Writing data to files is essential for creating logs, storing user input, and more.

### Overwriting an Existing File:

Opening a file in write mode (`"w"`) overwrites the existing content.

```
with open('example.txt', 'w') as file:
    file.write("This will overwrite existing content.")
```

### Appending to a File:

Opening a file in append mode (`"a"`) adds content to the end without altering existing data.

```
with open('example.txt', 'a') as file:
    file.write("\nThis line is appended.")
```

**Writing Multiple Lines:**

```
lines = ["First line.\n", "Second line.\n", "Third line.\n"]

with open('example.txt', 'w') as file:
    file.writelines(lines)
```

**Note: The `writelines()` method doesn't add newline characters automatically. Ensure each string ends with `\n` if needed.**

# *Creating a New File*

Creating new files can be done using different modes depending on your requirements.

## Using Write Mode (`"w"`):

```
with open('new_file.txt', 'w') as file:
    file.write("This is a new file.")
```

**Behavior:** Creates `new_file.txt` if it doesn't exist. If it exists, truncates its content.

Using Exclusive Creation Mode (`"x"`):

```
try:
    with open('exclusive_file.txt', 'x') as file:
        file.write("This file is created exclusively.")
except FileExistsError:
    print("File already exists.")
```

**Behavior:** Creates the file only if it doesn't exist. Raises `FileExistsError` if the file exists.
**Use Case:** Prevent accidental overwriting of existing files.

## *Checking if a File Exists*

Before performing operations like reading or writing, it's often necessary to check if a file exists.

### Using the `os` Module:

```
import os

file_path = 'example.txt'

if os.path.exists(file_path):
```

```
    print("File exists.")
else:
    print("File does not exist.")
```

**Note:** `os.path.exists()` returns `True` for both files and directories. To check specifically for files:

```
import os

file_path = 'example.txt'

if os.path.isfile(file_path):
    print("File exists and is a regular file.")
else:
    print("File does not exist or is not a regular file.")
```

**Using the `pathlib` Module:**

```
from pathlib import Path

file = Path('example.txt')

if file.exists():
    print("File exists.")
else:
    print("File does not exist.")
```

Checking if It's a File:

```
from pathlib import Path

file = Path('example.txt')

if file.is_file():
    print("File exists and is a regular file.")
else:
    print("File does not exist or is not a regular file.")
```

## *Deleting a File*

Removing unwanted files helps in maintaining a clean file system.

### Using the os Module:

```
import os

file_path = 'unwanted_file.txt'
```

```python
try:
    os.remove(file_path)
    print(f"{file_path} has been deleted.")
except FileNotFoundError:
    print("File does not exist.")
except PermissionError:
    print("Permission denied.")
```

Using the `pathlib` Module:

```python
from pathlib import Path

file = Path('unwanted_file.txt')

try:
    file.unlink()
    print(f"{file} has been deleted.")
except FileNotFoundError:
    print("File does not exist.")
except PermissionError:
    print("Permission denied.")
```

**Note:** `unlink()` is the `pathlib` equivalent of `os.remove()`.

# *Deleting a Folder*

Removing directories (folders) can be straightforward or require caution, especially if the directory contains files.

## Deleting an Empty Directory

### Using the `os` Module:

```
import os

folder_path = 'empty_folder'

try:
    os.rmdir(folder_path)
    print(f"{folder_path} has been deleted.")
except FileNotFoundError:
    print("Folder does not exist.")
except OSError:
    print("Folder is not empty or cannot be deleted.")
```

### **Using the `pathlib` Module:**

```
from pathlib import Path

folder = Path('empty_folder')
```

```python
try:
    folder.rmdir()
    print(f"{folder} has been deleted.")
except FileNotFoundError:
    print("Folder does not exist.")
except OSError:
    print("Folder is not empty or cannot be deleted.")
```

## Deleting a Non-Empty Directory

For directories containing files or subdirectories, use the `shutil` module.

### Using `shutil.rmtree`:

```python
import shutil

folder_path = 'non_empty_folder'

try:
    shutil.rmtree(folder_path)
    print(f"{folder_path} and all its contents have been deleted.")
except FileNotFoundError:
```

```
    print("Folder does not exist.")
except PermissionError:
    print("Permission denied.")
```

**Caution:** `shutil.rmtree()` permanently deletes the directory and all its contents. Use it carefully to avoid accidental data loss.

## *Handling Exceptions in File Operations*

File operations are prone to various exceptions, such as missing files, permission issues, or I/O errors. Proper exception handling ensures that your program can gracefully handle such scenarios.

| Exception | Description |
|---|---|
| `FileNotFoundError` | Raised when a file or directory is not found. |
| `PermissionError` | Raised when the operation lacks the necessary permissions. |
| `IsADirectoryError` | Raised when a directory operation is requested on a non-directory. |
| `EOFError` | Raised when the `end-of-file` condition is reached. |
| `IOError` | General I/O related errors. (In Python 3.x, `IOError` is an alias for `OSError`.) |

**Example: Handling File Not Found**

```
try:
    with open('non_existent_file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist.")
```

**Example: Handling Permission Denied**

```
try:
    with open('/root/secret_file.txt', 'r') as file:
        content = file.read()
except PermissionError:
    print("You do not have permission to read this file.")
```

## General Exception Handling

While it's good practice to handle specific exceptions, you can also catch general exceptions to prevent your program from crashing unexpectedly.

```
try:
    with open('example.txt', 'r') as file:
        content = file.read()
except Exception as e:
    print(f"An error occurred: {e}")
```

**Note:** Avoid using bare `except:` clauses as they can catch unexpected exceptions, making debugging harder.

# ***Practical Example***

## Example 1: Reading a Configuration File

**Scenario:** You have a configuration file `config.txt` containing key-value pairs.

**Content of `config.txt`:**

username=admin

```
password=secret
host=localhost
port=8080
```

Reading and Parsing the Configuration:

```
config = {}

with open('config.txt', 'r') as file:
    for line in file:
        line = line.strip()
        if line and not line.startswith('#'):  # Ignore empty lines and comments
            key, value = line.split('=', 1)
            config[key.strip()] = value.strip()

print(config)
```

**Output:**

{'username': 'admin', 'password': 'secret', 'host': 'localhost', 'port': '8080'}

## Example 2: Logging Events to a File

**Scenario:** You want to log application events with timestamps.

**Logging Function:**

```
from datetime import datetime

def log_event(message, log_file='app.log'):
    timestamp = datetime.now().strftime("%Y-%m-%d %H:
%M:%S")
    entry = f"[{timestamp}] {message}\n"
    with open(log_file, 'a') as file:
        file.write(entry)

# Usage
log_event("Application started.")
log_event("User logged in.")
```

**Content of `app.log` After Execution:**

```
[2024-10-08 12:30:45] Application started.
[2024-10-08 12:31:10] User logged in.
```

## Example 3: Processing a CSV File

**Scenario:** You have a CSV file `data.csv` and want to process its contents.

**Content of `data.csv`:**

```
Name,Age,Country
Alice,30,USA
Bob,25,Canada
Charlie,35,UK
```

Reading and Processing the CSV File:

```python
import csv

with open('data.csv', 'r', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        name = row['Name']
        age = int(row['Age'])
        country = row['Country']
        print(f"{name} is {age} years old and lives in {country}.")
```

Output:

```
Alice is 30 years old and lives in USA.
Bob is 25 years old and lives in Canada.
Charlie is 35 years old and lives in UK.
```

## Example 4: Creating and Writing to a New File

**Scenario:** You want to create a new file `report.txt` and write a summary.

```python
summary = """
Report Summary
--------------
Total Users: 150
Active Users: 120
New Signups Today: 5
"""

with open('report.txt', 'w') as file:
    file.write(summary)

print("Report created successfully.")
```

**Output:**
Report created successfully.

**Content of `report.txt`:**
Report Summary
--------------
Total Users: 150
Active Users: 120
New Signups Today: 5

### Example 5: Deleting a Temporary File

**Scenario:** After processing, you want to delete a temporary file `temp_data.txt`.

```
import os

temp_file = 'temp_data.txt'

if os.path.exists(temp_file):
    try:
        os.remove(temp_file)
        print(f"{temp_file} has been deleted.")
    except PermissionError:
        print("Permission denied. Cannot delete the file.")
else:
    print("Temporary file does not exist.")
```

**Output (if `temp_data.txt` exists):**

```
temp_data.txt has been deleted.
```

## *Exercises*

## Exercise 1: Counting Lines in a File

**Task:** Write a Python script that counts the number of lines in a given text file `sample.txt`.

```
def count_lines(file_path):
    try:
        with open(file_path, 'r') as file:
            return sum(1 for _ in file)
    except FileNotFoundError:
        print("File not found.")
        return 0

# Usage
line_count = count_lines('sample.txt')
print(f"Number of lines: {line_count}")
```

## Exercise 2: Copying a File

**Task:** Write a Python function that copies the contents of `s-ource.txt` to `destination.txt`.

**Solution:**

```
import shutil

def copy_file(source, destination):
    try:
```

```
        shutil.copy(source, destination)
        print(f"Copied {source} to {destination}.")
    except FileNotFoundError:
        print("Source file does not exist.")
    except PermissionError:
        print("Permission denied.")

# Usage
copy_file('source.txt', 'destination.txt')
```

## Exercise 3: Reading a File and Removing Blank Lines

**Task:** Write a Python script that reads `input.txt`, removes any blank lines, and writes the result to `output.txt`.

**Solution:**

```
def remove_blank_lines(input_file, output_file):
    with open(input_file, 'r') as infile, open(output_file, 'w') as outfile:
        for line in infile:
            if line.strip():  # Checks if the line is not empty or whitespace
                outfile.write(line)

# Usage
```

```
remove_blank_lines('input.txt', 'output.txt')
print("Blank lines removed.")
```

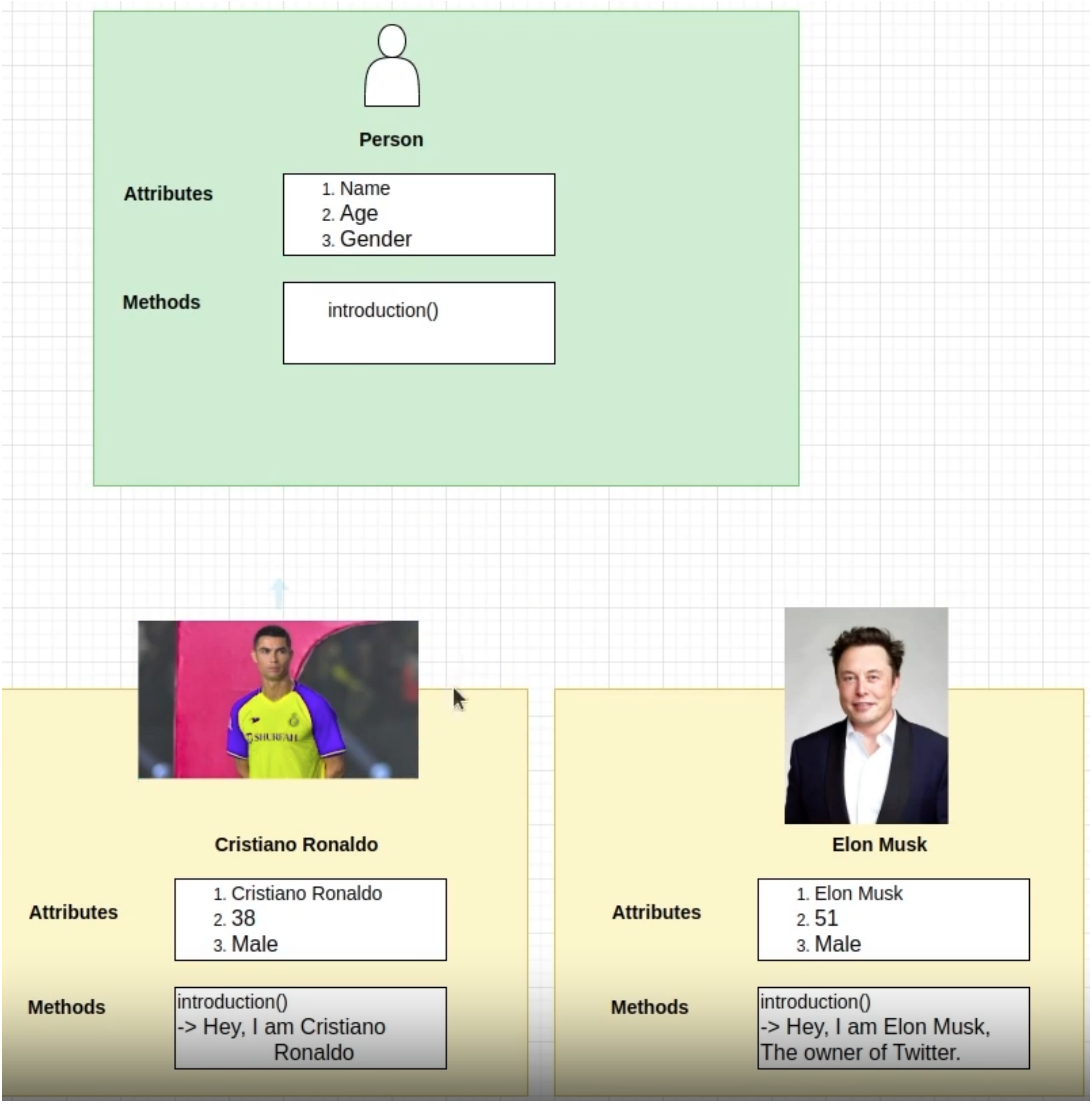
## Exercise 4: Merging Multiple Text Files

**Task:** Write a Python script that merges all `.txt` files in a directory into a single file `merged.txt`.

**Solution:**

```
from pathlib import Path

def merge_text_files(directory, output_file='merged.txt'):
    dir_path = Path(directory)
    with open(output_file, 'w') as outfile:
        for txt_file in dir_path.glob('*.txt'):
            with open(txt_file, 'r') as infile:
                outfile.write(infile.read() + '\n')  # Add a newline
between files
    print(f"All text files merged into {output_file}.")

# Usage
merge_text_files('.')
```

# *Object Oriented Programming*




Here, Person is a class and Cristiano Ronaldo and Elon Musk is an instance of class.

Class: Class is an user-define data type. It's structure of object
Object: Instances of a class. It is created using the class

definition as blueprint and it has its own unique identity, state,and behaviour.

**Object-Oriented Programming (OOP)** is a programming paradigm based on the concept of "objects", which can contain data and code:
• **Data** in the form of fields (attributes or properties).
• **Code** in the form of procedures (methods).

**Key Principles of OOP:**
1. **Encapsulation:** Bundling data and methods that operate on the data within one unit (class) and restricting access to some of the object's components.
2. **Inheritance:** Creating new classes from existing ones to promote code reusability.
3. **Polymorphism:** Allowing entities to be represented in multiple forms.
4. **Abstraction:** Hiding complex implementation details and showing only the essential features.
Python's OOP model is flexible and dynamic, enabling developers to build complex systems with ease.

# *Classes and Objects*

## Classes

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

Syntax:

```
class ClassName:
    """Docstring describing the class."""

    # Class attributes
    class_attribute = value

    # Constructor method
    def __init__(self, parameters):
        # Instance attributes
        self.instance_attribute = value

    # Methods
    def method_name(self, parameters):
        # Method body
        pass
```

**Example:**

```
class Dog:
    """A simple Dog class."""
```

```python
    # Class attribute
    species = "Canis familiaris"

    # Constructor
    def __init__(self, name, age):
        # Instance attributes
        self.name = name
        self.age = age

    # Method
    def bark(self):
        return f"{self.name} says woof!"
```

## Objects (Instances)

An **object** is an instance of a class. It encapsulates data and behavior defined by its class.

**Creating an Object:**

```python
# Instantiate the Dog class
my_dog = Dog("Buddy", 5)

# Accessing attributes
print(my_dog.name)   # Output: Buddy
print(my_dog.age)    # Output: 5
print(my_dog.species)  # Output: Canis familiaris
```

```python
# Calling methods
print(my_dog.bark())  # Output: Buddy says woof!
```

# ***Attributes and Methods***

## Attributes and Methods

### Attributes

Attributes are variables that belong to a class or its instances.

- **Class Attributes:** Shared across all instances of the class.
- **Instance Attributes:** Unique to each instance.

**Example:**

```python
class Car:
    # Class attribute
    wheels = 4

    def __init__(self, brand, model):
        # Instance attributes
        self.brand = brand
```

```python
        self.model = model

# Creating instances
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

print(car1.wheels)  # Output: 4
print(car2.wheels)  # Output: 4

print(car1.brand)   # Output: Toyota
print(car2.brand)   # Output: Honda
```

## Methods

Methods are functions defined within a class that describe the behaviors of the objects.

• **Instance Methods:** Operate on instance attributes.

• **Class Methods:** Operate on class attributes and are marked with the `@classmethod` decorator.

• **Static Methods:** Do not operate on class or instance attributes and are marked with the `@staticmethod` decorator.

**Example of Instance Method:**

```python
class Rectangle:
    def __init__(self, width, height):
```

```
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Usage
rect = Rectangle(5, 10)
print(rect.area())  # Output: 50
```

## *Encapsulation*

**Encapsulation** is the bundling of data and methods that operate on that data within one unit (class). It restricts direct access to some of an object's components, which is a means of preventing accidental interference and misuse.

### Private Attributes and Methods

In Python, **private** attributes and methods are indicated by a leading underscore (_) or double underscores (__). However, Python does not enforce strict access control; it's a convention to indicate intended privacy.

• **Single Underscore (_):** Indicates that the attribute or method is intended for internal use (protected).
• **Double Underscore (__):** Triggers name mangling to make it harder to access the attribute or method from outside the class

**Example:**
```
class Person:
    def __init__(self, name, age):
        self.name = name        # Public attribute
        self._age = age         # Protected attribute
        self.__salary = 50000   # Private attribute

    def get_salary(self):
        return self.__salary

    def __private_method(self):
        print("This is a private method.")

# Usage
p = Person("Alice", 30)
print(p.name)       # Output: Alice
print(p._age)       # Output: 30
# print(p.__salary)  # AttributeError

print(p.get_salary())  # Output: 50000
```

```
# Attempting to call private method
# p.__private_method()  # AttributeError

# Accessing via name mangling
print(p._Person__salary)        # Output: 50000
p._Person__private_method()      # Output: This is a private method.
```

**Note:** While name mangling makes it harder to access private attributes and methods, it doesn't make them truly private. It's a convention to discourage external access rather than enforce it.

## Benefits of Encapsulation

- **Data Hiding:** Protects object integrity by preventing external components from altering internal states.
- **Modularity:** Enhances code modularity by grouping related attributes and methods.
- **Maintainability:** Simplifies maintenance by localizing changes within classes.

Important Note:

```
# Mangling in Python is indeed a built-in feature designed to
```

avoid unwanted access or conflicts by renaming private attributes and methods.
# Accessing these mangled names is possible, but it's not recommended because it goes against the intended use of encapsulation.
# When we defined private attribute or method it changes to private method or attribute to
_className.__methodName() or
_className.__attribute_name

# *Inheritace*

## Inheritance

**Inheritance** allows a class (child or subclass) to inherit attributes and methods from another class (parent or superclass). It promotes code reusability and establishes a natural hierarchy between classes.

### Basic Inheritance

**Example:**

```
class Animal:
    def __init__(self, name):
        self.name = name
```

```
    def speak(self):
        return "Some generic sound."

# Subclass inheriting from Animal
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Subclass inheriting from Animal
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Usage
dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.name)      # Output: Buddy
print(dog.speak())   # Output: Woof!

print(cat.name)      # Output: Whiskers
print(cat.speak())   # Output: Meow!
```

## Using super()

The super() function allows a subclass to access methods and properties of its superclass.

**Example:**

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def drive(self):
        return f"{self.brand} is driving."

class Car(Vehicle):
    def __init__(self, brand, model):
        # Initialize attributes from the superclass
        super().__init__(brand)
        self.model = model

    def drive(self):
        # Call the superclass method
        base_drive = super().drive()
        return f"{base_drive} It's a {self.model}."

# Usage
car = Car("Toyota", "Corolla")
print(car.brand)       # Output: Toyota
print(car.model)       # Output: Corolla
print(car.drive())     # Output: Toyota is driving. It's a Corolla.
```

# *Multiple Inheritance*

## Multiple Inheritance

Python supports multiple inheritance, allowing a class to inherit from multiple superclasses.

**Example:**

```
class Flyer:
    def fly(self):
        return "Flying."

class Swimmer:
    def swim(self):
        return "Swimming."

class Duck(Flyer, Swimmer):
    def quack(self):
        return "Quack!"

# Usage
donald = Duck()
```

```
print(donald.fly())    # Output: Flying.
print(donald.swim())   # Output: Swimming.
print(donald.quack())  # Output: Quack!
```

**Caution:** Multiple inheritance can lead to complexity and the **Diamond Problem**, where the inheritance hierarchy forms a diamond shape. Python's **Method Resolution Order (MRO)** handles this by defining the order in which base classes are searched when executing a method.

# *Diamond Problem*

## The Diamond Problem and MRO

**Example:**

```
class A:
    def method(self):
        print("Method in A")

class B(A):
    def method(self):
        print("Method in B")

class C(A):
```

```
    def method(self):
        print("Method in C")

class D(B, C):
    pass

# Usage
d = D()
d.method()  # Output: Method in B
print(D.mro())
# Output: [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

In this example, class `D` inherits from both `B` and `C`. When `d.method()` is called, Python follows the MRO and finds `method` in class `B` before class `C`, hence `Method in B` is printed.

**Understanding MRO:**

Use the `.mro()` method or `help(ClassName)` to view the MRO.

```
print(D.mro())
# Output: [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

Example:

```
# class A:
#     def __init__(self):
#         print("Class A")
#
# class B(A):
#     def __init__(self):
#         print("Class B")
#         A.__init__(self)
# class C(B):
#     def __init__(self):
#         print("Class C")
#         B.__init__(self)
#
# class D(C,B):
#     def __init__(self):
#         print("Class D")
#         C.__init__(self)
#         B.__init__(self)
#
#
# class E(D,C):
#     def __init__(self):
#         print("Class D")
#         D.__init__(self)
#         C.__init__(self)
# d = E()
```

# Diamond problem occurs here we see D, C, B. and A are called multiple time. So solve this problem we can use super() mehtod

```
# class A:
#     def __init__(self):
#         print("A")
#
# class B(A):
#     def __init__(self):
#         print("B")
#         super().__init__()
#
# class C(B):
#     def __init__(self):
#         print("C")
#         super().__init__()
#
# class D(B, C):
#     def __init__(self):
#         print("D")
#         super().__init__()
#
# d = D()
# print(D.__mro__)
```

# Cause of Error

# First Call to D.__init__():
#
# When you create an instance of D, the constructor of D is called first. This is correct.
# Call to super().__init__() in D:
#
# The super() function in D tries to call the next class in the MRO (Method Resolution Order), which is B.
# Call to super().__init__() in B:
#
# Inside B.__init__(), super() is called, which normally would refer to A because B inherits from A.
# Expected Behavior:
#
# However, in the case of D(B, C), due to multiple inheritance, the next class to be called (after B) should be C, not Aâ€”because C is listed in D's inheritance declaration.
# The Problem:
#
# The real issue arises from the fact that both B and C inherit from A in some form, which causes a conflict in the inheritance chain. Python's MRO can't decide how to handle this situation consistently because C should also call B according to your setup, but B already calls A.
# The MRO Error:
#
# Python gets confused in resolving this order, and it throws

an MRO inconsistency error. The conflict arises because Python can't decide whether A should be called from B first or from C. So, the MRO is ambiguous, which is why the error occurs.

```
class A:
    def __init__(self):
        print("A")

class B(A):
    def __init__(self):
        print("B")
        super().__init__()

class C(B):
    def __init__(self):
        print("C")
        super().__init__()

class D(C, B):
    def __init__(self):
        print("D")
        super().__init__()

d = D()
print(D.__mro__)
```

## *Single Inheritance*

```
# In single inheritance, a class inherits from only one base class.
class Animal:
    def speak(self):
        print("Animal an speak")

class Cat(Animal):
    def meow(self):
        print("Cat say something to you! Meow")

cat = Cat()
cat.meow()
```

## *Multilevel Inheritance*

```
# In multilevel inheritance, a class inherits from another class, and then a third class inherits from the second class.
```

```
class Shape:
    def __init__(self, height, width):
        self.__height = height
        self.__width = width

    def is_shape(self):
        if self.__height > 0 and self.__width > 0:
            print("it's a shape")


class Rectangle(Shape):
    def __init__(self, height, width):
        super().__init__(height,width)

class Circles(Rectangle):
    def __init__(self, height, width, radius):
        self.radius = radius
        super().__init__(height, width)

circle = Circles(1,4,5)
rect = Rectangle(1,5)

rect.is_shape()
circle.is_shape()
```

## *Hierarchical Inheritance*

```
# In hierarchical inheritance, multiple classes inherit from a common base class.
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")


# Derived class1


class Boy(Parent):
    def func2(self):
        print("This function is in Boy.")


# Derivied class2
class Girl(Parent):
    def func3(self):
        print("This function is in Girl.")
```

```python
# Driver's code
object1 = Boy()
object2 = Girl()

object1.func1()
object1.func2()

object2.func1()
object2.func3()
```

## *Hybrid Inheritance*

```python
# Combination of hierarchial and multiple inheritance
class A:
    def method1(self):
        print("this is method 1")


class B(A):
    def method2(self):
        print("this is method 2")


class C(A):
```

```python
    def method3(self):
        print("this is method 3")


class D(B, C):
    def method4(self):
        print("this is method 4")


obj1 = D()
obj1.method1()
obj1.method2()
obj1.method3()
obj1.method4()
```

## *Polymorphism*

**Polymorphism** allows objects of different classes to be treated as objects of a common superclass. It enables methods to perform different tasks based on the object it is acting upon.

## Duck Typing

Python follows the **Duck Typing** philosophy: "If it walks like a duck and quacks like a duck, it's a duck." This means that the type or class of an object is less important than the methods it defines.

Example:

```
class Bird:
    def fly(self):
        return "Flying."

class Airplane:
    def fly(self):
        return "Airplane is flying."

def make_it_fly(flyable):
    print(flyable.fly())

# Usage
bird = Bird()
airplane = Airplane()

make_it_fly(bird)       # Output: Flying.
make_it_fly(airplane)  # Output: Airplane is flying.
```

**Explanation:** Both `Bird` and `Airplane` have a `fly` method. The `make_it_fly` function doesn't care about the object's class, only that it has a `fly` method.

## Operator Overloading

Polymorphism also manifests in **operator overloading**, allowing custom behavior for built-in operators based on object types.

**Example:**

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Overload the '+' operator
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # Represent the object as a string
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

# Usage
v1 = Vector(2, 4)
v2 = Vector(5, -2)
v3 = v1 + v2

print(v3)  # Output: Vector(7, 2)
```

# *Abstraction*

**Abstraction** involves hiding complex implementation details and exposing only the essential features of an object. It allows developers to focus on what an object does rather than how it does it.

## Abstract Base Classes (ABC)

Python provides the abc module to define **abstract base classes**, which cannot be instantiated and can enforce that derived classes implement specific methods.

**Example:**

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

```
    @abstractmethod
    def perimeter(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.1416 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.1416 * self.radius

# Usage
# shape = Shape()  # TypeError: Can't instantiate abstract class Shape with abstract methods area, perimeter

circle = Circle(5)
print(circle.area())       # Output: 78.54
print(circle.perimeter())  # Output: 31.416
```

**Explanation:** The `Shape` class defines abstract methods `area` and `perimeter`. Any subclass of `Shape` must implement these methods. Attempting to instantiate `Shape` directly results in a `TypeError`.

## Benefits of Abstraction

• **Simplifies Complexity:** Hides intricate details, making the interface easier to use.
• **Promotes Code Reusability:** Encourages the use of general interfaces that can be implemented by multiple classes.
• **Enhances Maintainability:** Changes in implementation do not affect code that relies on the abstract interface

# *Special (Magic) Methods*

**Special methods**, also known as **magic methods**, are predefined methods in Python that begin and end with double underscores (__). They allow objects to implement and interact with built-in operations and functions.

## Common Magic Methods

**Special methods**, also known as **magic methods**, are predefined methods in Python that begin and end with double underscores (__). They allow objects to implement and interact with built-in operations and functions.

**1. `__str__` and `__repr__`**

- **`__str__`:** Used by the `str()` function and the `print` statement to create a readable string representation of an object.
- **`__repr__`:** Used by the `repr()` function and interactive shells to create an unambiguous string representation of an object, ideally one that could be used to recreate the object.

Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Point({self.x}, {self.y})"

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

# Usage
p = Point(3, 4)
print(p)          # Output: Point(3, 4)
print(str(p))     # Output: Point(3, 4)
print(repr(p))    # Output: Point(x=3, y=4)
```

2. Operator Overloading with `__add__`

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        return NotImplemented

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

# Usage
v1 = Vector(2, 3)
v2 = Vector(5, 7)
v3 = v1 + v2
print(v3)  # Output: Vector(7, 10)
```

3. Making an Object Callable with `__call__`

```
class Multiplier:
    def __init__(self, factor):
        self.factor = factor
```

```
    def __call__(self, number):
        return self.factor * number

# Usage
double = Multiplier(2)
triple = Multiplier(3)

print(double(5))  # Output: 10
print(triple(5))  # Output: 15
```

4. Using `__len__`

```
class CustomList:
    def __init__(self, elements):
        self.elements = elements

    def __len__(self):
        return len(self.elements)

# Usage
cl = CustomList([1, 2, 3, 4, 5])
print(len(cl))  # Output: 5
```

5. Context Managers with `__enter__` and `__exit__`

```
class ManagedFile:
    def __init__(self, filename):
        self.filename = filename
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, 'w')
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()

# Usage
with ManagedFile('test.txt') as f:
    f.write("Hello, World!")

# 'test.txt' is automatically closed after the block
```

# *Class vs. Instance Variables*

## Class vs. Instance Variables

Understanding the difference between **class variables** and **instance variables** is crucial for effective OOP in Python.

### Class Variables

- **Definition:** Variables that are shared among all instances of a class.
- **Usage:** To store data that should be consistent across all instances.
- **Access:** Via the class name or via instances.

**Example:**

```
class Employee:
    # Class variable
    company = "ABC Corp"

    def __init__(self, name):
        self.name = name  # Instance variable

# Usage
emp1 = Employee("Alice")
emp2 = Employee("Bob")
```

```python
print(emp1.company)  # Output: ABC Corp
print(emp2.company)  # Output: ABC Corp

# Modifying class variable via the class
Employee.company = "XYZ Inc"
print(emp1.company)  # Output: XYZ Inc
print(emp2.company)  # Output: XYZ Inc
```

## Instance Variables

• **Definition:** Variables that are unique to each instance of a class.
• **Usage:** To store data that varies between instances.
• **Access:** Only via instances.

**Example:**

```python
class Employee:
    def __init__(self, name, salary):
        self.name = name      # Instance variable
        self.salary = salary  # Instance variable

# Usage
emp1 = Employee("Alice", 70000)
emp2 = Employee("Bob", 80000)

print(emp1.name)    # Output: Alice
```

```
print(emp2.name)    # Output: Bob

print(emp1.salary)  # Output: 70000
print(emp2.salary)  # Output: 80000
```

## Important Notes

• **Class Variable Overridden by Instance Variable:** If an instance variable shares the same name as a class variable, the instance variable takes precedence.

```
class Employee:
    company = "ABC Corp"

    def __init__(self, name):
        self.company = "Personal Project"  # Overrides class variable for this instance

emp = Employee("Charlie")
print(emp.company)      # Output: Personal Project
print(Employee.company)  # Output: ABC Corp
```

**Accessing Class Variables:** Access class variables using the class name to avoid confusion with instance variables.

```
class Employee:
```

```
    company = "ABC Corp"

    def __init__(self, name):
        self.name = name

emp = Employee("Diana")
print(Employee.company)  # Output: ABC Corp
print(emp.company)       # Output: ABC Corp
```

# ***Static and Class Methods***

Python provides two special types of methods in classes: **static methods** and **class methods**. These methods are not bound to instances but have specific use cases.

## Class Methods

- **Definition:** Methods that receive the class as the first argument, typically named `cls`.
- **Decorator:** `@classmethod`
- **Use Cases:** Factory methods, modifying class state.

**Example:**

```
class Employee:
```

```python
    company = "ABC Corp"

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    @classmethod
    def change_company(cls, new_company):
        cls.company = new_company

    @classmethod
    def from_string(cls, emp_str):
        name, salary = emp_str.split('-')
        return cls(name, int(salary))

# Usage
emp1 = Employee("Alice", 70000)
print(emp1.company)  # Output: ABC Corp

# Changing company using class method
Employee.change_company("XYZ Inc")
print(emp1.company)  # Output: XYZ Inc

# Creating an instance using a factory class method
emp_str = "Bob-80000"
emp2 = Employee.from_string(emp_str)
print(emp2.name)    # Output: Bob
```

```
print(emp2.salary)  # Output: 80000
print(emp2.company) # Output: XYZ Inc
```

## Static Methods

• **Definition:** Methods that do not receive an implicit first argument (neither `self` nor `cls`).
• **Decorator:** `@staticmethod`
• **Use Cases:** Utility functions that perform a task in isolation.

**Example:**

```
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def multiply(a, b):
        return a * b

# Usage
print(MathOperations.add(5, 3))        # Output: 8
print(MathOperations.multiply(5, 3))   # Output: 15

# Can also be called via an instance
math_op = MathOperations()
```

print(math_op.add(2, 4))            # Output: 6

**Key Differences**

| Feature | Instance Methods | Class Methods | Static Methods |
|---|---|---|---|
| **First Parameter** | `self` | `cls` | None |
| **Access** | Via instances | Via class or instances | Via class or instances |
| **Use Cases** | Manipulate instance data | Manipulate class data or create instances | Utility functions |

# *Properties*

**Properties** allow you to manage the access to instance attributes by defining methods that get, set, or delete their values. They provide a way to add logic around getting or setting an attribute, enforcing encapsulation.

## Using `@property` Decorator

The `@property` decorator transforms a method into a "getter" for a read-only attribute.

**Example:**

class Person:

```python
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"

# Usage
p = Person("John", "Doe")
print(p.full_name)  # Output: John Doe
# p.full_name = "Jane Smith"  # AttributeError: can't set attribute
```

## Setting a Property with `@<property>.setter`

To allow setting the value, define a setter method using `@<property>.setter`.

**Example:**

```python
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
```

```
    def full_name(self):
        return f"{self.first_name} {self.last_name}"

    @full_name.setter
    def full_name(self, name):
        first, last = name.split(' ', 1)
        self.first_name = first
        self.last_name = last

# Usage
p = Person("John", "Doe")
print(p.full_name)  # Output: John Doe

p.full_name = "Jane Smith"
print(p.first_name)  # Output: Jane
print(p.last_name)   # Output: Smith
```

## Deleting a Property with `@<property>.deleter`

You can also define a deleter method to handle attribute deletion.

**Example:**

```
# Deleting a Property with @<property>.deleter
# You can also define a deleter method to handle attribute deletion.
```

```python
class Person2:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'

    @full_name.setter
    def full_name(self, name):
        first_name, last_name = name.split(' ', 1)
        self.first_name = first_name
        self.last_name = last_name

    @full_name.deleter
    def full_name(self):
        print("Delete Full Name")
        del self.first_name
        del self.last_name


#Usage
p = Person2("Abu", "Huraira")
print(p.full_name)
```

```python
p.full_name = "Abu Abdullah"
print(p.full_name)

# Attempting to access attributes after deletion
del p.full_name
# print(p.first_name) # AttributeError
# print(p.full_name) # AttributeError
```

### Benefits of Using Properties

- **Encapsulation:** Control access to attributes, enforcing validation or other logic.
- **Flexibility:** Change internal implementation without altering the external interface.
- **Readability:** Access attributes like regular properties without explicit getter and setter calls.

# *Practical Examples*

### Example 1: Bank Account Management

**Scenario:** Create a `BankAccount` class to manage bank accounts with functionalities like deposit, withdrawal, and displaying balance.

```
class BankAccount:
    account_number_counter = 1000  # Class variable to
assign unique account numbers

    def __init__(self, owner, balance=0):
        self.owner = owner            # Instance variable
        self.balance = balance        # Instance variable
        self.account_number =
BankAccount.account_number_counter
        BankAccount.account_number_counter += 1

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount}. New balance: $
{self.balance}.")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew ${amount}. New balance: $
{self.balance}.")
```

```
        elif amount > self.balance:
            print("Insufficient funds.")
        else:
            print("Withdrawal amount must be positive.")

    def display_balance(self):
        print(f"Account {self.account_number} owned by
{self.owner} has balance: ${self.balance}.")

# Usage
acc1 = BankAccount("Alice", 500)
acc2 = BankAccount("Bob")

acc1.deposit(200)       # Output: Deposited $200. New
balance: $700.
acc1.withdraw(100)      # Output: Withdrew $100. New
balance: $600.
acc1.display_balance()  # Output: Account 1000 owned by
Alice has balance: $600.

acc2.display_balance()  # Output: Account 1001 owned by
Bob has balance: $0.
acc2.withdraw(50)       # Output: Insufficient funds.
acc2.deposit(300)       # Output: Deposited $300. New
balance: $300.
acc2.display_balance()  # Output: Account 1001 owned by
Bob has balance: $300.
```

## Example 2: Library Management System

**Scenario:** Create classes to manage books and library members, allowing members to borrow and return books.

```
class Book:

    def __init__(self, title, author, no_of_book=1):
        self.title = title
        self.author = author
        self.no_of_book = no_of_book

    def borrow(self):
        if self.no_of_book > 0:
            self.no_of_book -= 1
            print(f"Book '{self.title}' has been borrowed.")
            return True
        else:
            print(f"There is no available  Item to Book '{self.title}'. Please Try After few days.")
            return False
    def return_book(self):
        self.no_of_book += 1
        print(f"Book '{self.title}' has been returned.")
```

```
class Member:
    def __init__(self, name):
        self.name = name
        self.borrowed_book = []

    def borrow_book(self, book):
        if book.title not in self.borrowed_book and book.borrow
():
            self.borrowed_book.append(book.title)
            print(f"{self.name} borrowed '{book.title}'.")
        elif book.title in self.borrowed_book:
            print(f"{self.name} cannot borrow '{book.title}' as it is
already borrowed.")
        else:
            print(f"{self.name} does not have '{book.title}'
borrowed.")

    def return_book(self, book):
        if book.title in self.borrowed_book:
            book.return_book()
            self.borrowed_book.remove(book.title)
            print(f"{self.name} returned '{book.title}'.")
        else:
            print(f"{self.name} does not have '{book.title}'
borrowed.")

book1 = Book("1984", "George Orwell",2)
```

```
book2 = Book("To Kill a Mockingbird", "Harper Lee")

member1 = Member("Alice")
member2 = Member("Bob")
member3 = Member("Abdullah")

member1.borrow_book(book1)
print()

member2.borrow_book(book1)
print()

member3.borrow_book(book1)
print()

member2.return_book(book1)
print()

member3.borrow_book(book1)
print()

member2.borrow_book(book2)
```

Output:

```
Book '1984' has been borrowed.
Alice borrowed '1984'.

Book '1984' has been borrowed.
Bob borrowed '1984'.

There is no available  Item to Book '1984'. Please Try After few days.
Abdullah does not have '1984' borrowed.

Book '1984' has been returned.
Bob returned '1984'.

Book '1984' has been borrowed.
Abdullah borrowed '1984'.

Book 'To Kill a Mockingbird' has been borrowed.
Bob borrowed 'To Kill a Mockingbird'.
```

## Example 3: Employee Management with Inheritance

**Scenario:** Create a base `Employee` class and derived classes like `Developer` and `Manager`, each with specific attributes and methods.