Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming approach that organizes code using objects and classes instead of only functions and procedures. In simple words, OOP allows us to design programs by modeling real-world entities like students, cars, bank accounts, employees, etc. It makes programs more organized, reusable, and easier to maintain.

In Python, OOP is one of the most important programming paradigms. It helps in breaking large problems into smaller parts using classes and objects.

1.Class and Object

A **class** is a blueprint or template used to create objects. It defines what properties (variables) and behaviors (methods/functions) an object will have.

An **object** is an instance of a class. When we create an object, memory is allocated for it, and it can use the properties and methods defined in the class.

For example, if "Student" is a class, then "Rahul" and "Anita" can be objects created from that class.

Classes help in grouping related data and functions together. Objects help in representing real-world things inside a program.

Code:

```
class Student:
    def greet(self):
        print("Hello, I am a student.")


# Creating object
s1 = Student()
s1.greet()
```

Output:

```
Hello, I am a student.
```

2.Constructor (init Method)

A constructor is a special method in Python that is automatically executed when an object is created. In Python, the constructor method is written as __init__().

The main purpose of a constructor is to initialize the data members (variables) of the class.

When we create an object and pass values, the constructor assigns those values to the object.

Code:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)


s1 = Student("Susmitha", 20)
s1.display()
```

Output:

```
Name: Susmitha
Age: 20
```

3.Instance Variables and Instance Methods

Instance variables are variables that belong to a particular object. They are usually defined inside the constructor using the keyword self.

Instance methods are functions defined inside a class that work with instance variables. They always use self as the first parameter.

Each object has its own copy of instance variables, which means changing one object’s data will not affect another object.

Code:

```
lass Car:
    def __init__(self, brand):
        self.brand = brand  # instance variable

    def show_brand(self):  # instance method
        print("Car brand is", self.brand)

c1 = Car("BMW")
c1.show_brand()
```

Output:

```
Car brand is BMW
```

4.Class Variables

Class variables are shared among all objects of a class. They are defined directly inside the class but outside any method.

If a value should be common for all objects, we use a class variable.

For example, if all students belong to the same college, the college name can be stored as a class variable.

```
class College:
    college_name = "ABC College"  # class variable

    def __init__(self, name):
        self.name = name

    def display(self):
        print(self.name, "studies at", College.college_name)
```

```
s1 = College("Rahul")
s2 = College("Anita")

s1.display()
s2.display()
```

Output:

```
Rahul studies at ABC College
Anita studies at ABC College
```

5.Types of Methods in Python

Python supports three types of methods inside a class:

1. Instance Methods
2. Class Methods
3. Static Methods

Instance methods work with object data.
Class methods work with class-level data and use @classmethod decorator.
Static methods do not depend on class or object data and use @staticmethod decorator.

Code:

```
class Demo:
    count = 0

    def __init__(self):
        Demo.count += 1

    # Instance Method
    def show_instance(self):
        print("This is instance method.")
    # Class Method
    @classmethod
    def show_count(cls):
        print("Total objects created:", cls.count)

    # Static Method
    @staticmethod
    def greet():
        print("Hello from static method.")

d1 = Demo()
d2 = Demo()

d1.show_instance()
Demo.show_count()
Demo.greet()
```

Output:

```
This is instance method.
Total objects created: 2
Hello from static method.
```

6.The Four Pillars of OOP

The most important concepts in OOP are called the four pillars:

1. Encapsulation
2. Abstraction
3. Inheritance

4. Polymorphism

1. Encapsulation

Encapsulation means wrapping data and methods together inside a class and restricting direct access to some data.

In Python, we can make variables private by using double underscores (__). This prevents external code from directly modifying important data.

Encapsulation helps in:

- Protecting data
- Improving security
- Controlling how data is accessed

It ensures data integrity by allowing access only through methods.

Code:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # private variable

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

b1 = BankAccount(1000)
b1.deposit(500)
print("Balance:", b1.get_balance())
```

Output:

```
Balance: 1500
```

2. Abstraction

Abstraction means hiding internal implementation details and showing only essential features.

For example, when we use a car, we press the accelerator to move forward. We do not need to know how the engine works internally.

In Python, abstraction is implemented using abstract classes from the abc module.

Abstraction helps in:

- Reducing complexity
- Improving code clarity
- Focusing only on important features

Code:

```
from abc import ABC, abstractmethod


class Shape(ABC):
    @abstractmethod
    def area(self):
        pass


class Square(Shape):
    def __init__(self, side):
        self.side = side


    def area(self):
        print("Area:", self.side * self.side)


s = Square(4)
```

```
s.area()
```

Output:

```
Area: 16
```

3. Inheritance

Inheritance allows one class to acquire properties and methods from another class.

The class that gives properties is called the parent (or superclass), and the class that receives properties is called the child (or subclass).

Inheritance helps in:

- Code reusability
- Reducing duplication
- Creating hierarchical relationships

There are different types of inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

Inheritance allows the child class to override parent methods if needed.

Code:

```
class Animal:
    def speak(self):
        print("Animal makes sound")


class Dog(Animal):
    def bark(self):
        print("Dog barks")
```

```
d = Dog()
d.speak()
d.bark()
```

Output:

```
Animal makes sound
Dog barks
```

4. Polymorphism

Polymorphism means "many forms." It allows the same method name to behave differently in different situations.

For example, a function named sound() can produce different outputs for Dog, Cat, or Bird.

Polymorphism can be achieved using:

- Method overriding
- Operator overloading

It increases flexibility and allows writing generic code.

Code:

```
class Animal:
    def sound(self):
        print("Animal sound")


class Dog(Animal):
    def sound(self):
        print("Dog barks")


d = Dog()
d.sound()
```

Output:

Dog barks

7.Method Overriding

Method overriding happens when a child class defines a method with the same name as a method in the parent class.

The child class method replaces the parent class method.

This allows customization of inherited behavior.

8.Operator Overloading

Operator overloading allows us to redefine how operators like +, -, *, etc., behave for objects.

This is done using special methods like:

- __add__
- __sub__
- __mul__

It allows objects to behave like built-in data types.

Code:

```
class Number:
    def __init__(self, value):
        self.value = value


    def __add__(self, other):
        return self.value + other.value


n1 = Number(10)

n2 = Number(20)


print(n1 + n2)
```

Output:

```
30
```

9. **Magic (Dunder) Methods**

Magic methods are special methods that begin and end with double underscores.

Examples include:

- __init__() – constructor
- __str__() – string representation
- __len__() – length of object
- __add__() – addition operator

These methods allow customization of object behavior.

Code:

```
class Person:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"Person name is {self.name}"

p = Person("Susmitha")
print(p)
```

Output:

```
Person name is Susmitha
```

10. Access Modifiers in Python

Python has three types of access levels:

1. Public – accessible everywhere
2. Protected – indicated by single underscore (_)
3. Private – indicated by double underscore (__)

Although Python does not strictly enforce private access like some languages, it follows naming conventions.

Code:

```
class Test:
    def __init__(self):
        self.public = "Public"
        self._protected = "Protected"
        self.__private = "Private"


t = Test()
print(t.public)
print(t._protected)
```

Output:

```
Public
Protected
```

11.Composition

Composition means a class contains objects of another class.

It represents a "has-a" relationship.

For example:
A Car has an Engine.

Composition helps in building complex systems by combining small classes.

Code:

```
class Engine:
    def start(self):
        print("Engine started")
```

```
class Car:
    def __init__(self):
        self.engine = Engine()

    def start_car(self):
        self.engine.start()

c = Car()
c.start_car()
```

Output:

```
Engine started
```

12. Aggregation

Aggregation is similar to composition but weaker.

In aggregation, one class uses another class, but both can exist independently.

For example:
A Company has Employees, but Employees can exist without the Company.

Code:

```
class Employee:
    def __init__(self, name):
        self.name = name

class Company:
    def __init__(self, emp):
        self.emp = emp

e = Employee("Rahul")
```

```
c = Company(e)


print("Employee working:", c.emp.name)
```

Output:

```
Employee working: Rahul
```

13.Method Resolution Order (MRO)

When multiple inheritance is used, Python needs to decide which method to execute first.

MRO defines the order in which base classes are searched.

It prevents confusion in complex inheritance structures.

Code:

Method Resolution Order (MRO)

```
class A:
    pass
class B(A):
    pass
print(B.__mro__)
```

Output:

```
(<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Python Datatypes

Datatypes are classifications that specify which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error. Python is a dynamically typed language, meaning you don't have to declare the type of a variable when you create it; the interpreter infers it.

1. Numeric Types (Integers, Floats, Complex Numbers)

These represent numerical values.

- **Integers (int)**: Whole numbers, positive or negative, without decimals. E.g., 5, -100.

- **Floating-point numbers (float)**: Real numbers, with a decimal point. E.g., 3.14, -0.5.
- **Complex numbers (complex)**: Numbers with a real and imaginary part. E.g., 2 + 3j.

```
# Integers
int_var = 10
print(f"Integer: {int_var}, Type: {type(int_var)}")

# Floats
float_var = 10.5
print(f"Float: {float_var}, Type: {type(float_var)}")

# Complex Numbers
complex_var = 2 + 3j
print(f"Complex: {complex_var}, Type: {type(complex_var)}")

# Why use them?
# - Integers are used for counting, indexing, and discrete quantities.
# - Floats are used for measurements, calculations requiring precision, and continuous
quantities.
# - Complex numbers are used in advanced mathematical and engineering contexts.
```

```
Integer: 10, Type: <class 'int'>
Float: 10.5, Type: <class 'float'>
Complex: (2+3j), Type: <class 'complex'>
```

2. Strings (str)

Strings are sequences of characters, used for storing text. They are immutable, meaning once created, their content cannot be changed. Strings can be enclosed in single quotes ('...'), double quotes ("..."), or triple quotes ('''...''' or """...""") for multi-line strings.

```
# Single-line string
str_var1 = 'Hello, Python!'
print(f"String 1: {str_var1}, Type: {type(str_var1)}")

# Multi-line string
str_var2 = """This is a
multi-line string."""
print(f"String 2:\n{str_var2}, Type: {type(str_var2)}")

# String concatenation
full_string = str_var1 + " How are you?"
print(f"Concatenated String: {full_string}")

# String methods (e.g., upper, lower, replace)
print(f"Uppercase: {str_var1.upper()}")

# Why use them?
# - Essential for handling any kind of text data, such as names, messages, file paths, etc.
```

```
String 1: Hello, Python!, Type: <class 'str'>
String 2:
This is a
multi-line string., Type: <class 'str'>
Concatenated String: Hello, Python! How are you?
Uppercase: HELLO, PYTHON!
```

3. Booleans (bool)

Booleans represent truth values: True or False. They are fundamental for control flow (if/else statements, loops) and logical operations.

```
bool_true = True
bool_false = False
print(f"Boolean True: {bool_true}, Type: {type(bool_true)}")
print(f"Boolean False: {bool_false}, Type: {type(bool_false)}")

# Logical operations
print(f"True and False: {bool_true and bool_false}")
print(f"True or False: {bool_true or bool_false}")

# Why use them?
# - Used for decision-making, conditional logic, and representing states (e.g., 'on'/'off', 'active'/'inactive').
```

```
Boolean True: True, Type: <class 'bool'>
Boolean False: False, Type: <class 'bool'>
True and False: False
True or False: True
```

4. Lists (list)

Lists are ordered, mutable sequences of items. They can contain elements of different datatypes. Lists are defined using square brackets [].

```
list_var = [1, 'apple', 3.14, True]
print(f"List: {list_var}, Type: {type(list_var)}")

# Accessing elements (0-indexed)
```

```
print(f"First element: {list_var[0]}")


# Slicing
print(f"Slice (elements 1 and 2): {list_var[1:3]}")


# Modifying elements
list_var[1] = 'orange'
print(f"Modified list: {list_var}")


# Adding elements
list_var.append(False)
print(f"List after append: {list_var}")


# Why use them?
# - Ideal for collections of items that might change (e.g., a shopping cart, a list of student names that can be added/removed).
```

```
List: [1, 'apple', 3.14, True], Type: <class 'list'>
First element: 1
Slice (elements 1 and 2): ['apple', 3.14]
Modified list: [1, 'orange', 3.14, True]
List after append: [1, 'orange', 3.14, True, False]
```

5. Tuples (tuple)

Tuples are ordered, immutable sequences of items. Like lists, they can contain elements of different datatypes, but once a tuple is created, its contents cannot be changed. Tuples are defined using parentheses ().

```
tuple_var = (1, 'banana', 2.71, False)
```

```
print(f"Tuple: {tuple_var}, Type: {type(tuple_var)}")

# Accessing elements
print(f"Second element: {tuple_var[1]}")

# Trying to modify (will cause an error)
# tuple_var[1] = 'grape' # Uncommenting this line will raise a TypeError

# Why use them?
# - Used for collections of items that should not change (e.g., geographical coordinates, database records).
# - Can be used as keys in dictionaries (unlike lists) because they are immutable.
```

```
Tuple: (1, 'banana', 2.71, False), Type: <class 'tuple'>
Second element: banana
```

6. Dictionaries (dict)

Dictionaries are unordered collections of key-value pairs. Each key must be unique and immutable, while values can be of any datatype and can be changed. Dictionaries are defined using curly braces {}.

```
dict_var = {'name': 'Alice', 'age': 30, 'city': 'New York'}
print(f"Dictionary: {dict_var}, Type: {type(dict_var)}")

# Accessing values by key
print(f"Name: {dict_var['name']}")

# Modifying a value
dict_var['age'] = 31
```

```
print(f"Modified age: {dict_var['age']}")

# Adding a new key-value pair
dict_var['occupation'] = 'Engineer'
print(f"Dictionary after adding new key: {dict_var}")

# Why use them?
# - Excellent for storing structured data where you need to associate a value with a specific key (e.g., user profiles, configuration settings).
```

```
Dictionary: {'name': 'Alice', 'age': 30, 'city': 'New York'}, Type: <class 'dict'>
Name: Alice
Modified age: 31
Dictionary after adding new key: {'name': 'Alice', 'age': 31, 'city': 'New York', 'occupation': 'Engineer'}
```

7. Sets (set)

Sets are unordered collections of unique elements. They are mutable. Sets are defined using curly braces {} or the set() constructor. Duplicate elements are automatically removed.

```
set_var = {1, 2, 3, 2, 4}
print(f"Set: {set_var}, Type: {type(set_var)}") # Notice the duplicate '2' is removed

# Adding elements
set_var.add(5)
print(f"Set after adding 5: {set_var}")

# Removing elements
set_var.remove(1)
```

```
print(f"Set after removing 1: {set_var}")


# Set operations
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(f"Union: {set1.union(set2)}")
print(f"Intersection: {set1.intersection(set2)}")


# Why use them?
# - Useful for storing unique items, performing mathematical set operations (union, intersection, difference), and quickly checking for membership.
```

```
Set: {1, 2, 3, 4}, Type: <class 'set'>
Set after adding 5: {1, 2, 3, 4, 5}
Set after removing 1: {2, 3, 4, 5}
Union: {1, 2, 3, 4, 5}
Intersection: {3}
```

4. Lists (list)

Lists are ordered, mutable sequences of items. They can contain elements of different datatypes. Lists are defined using square brackets [].

Why use them?

- Ideal for collections of items that might change (e.g., a shopping cart, a list of student names that can be added/removed).

```
list_var = [1, 'apple', 3.14, True]
print(f"List: {list_var}, Type: {type(list_var)}")


# Accessing elements (0-indexed)
```

```
print(f"First element: {list_var[0]}")

# Slicing
print(f"Slice (elements 1 and 2): {list_var[1:3]}")

# Modifying elements
list_var[1] = 'orange'
print(f"Modified list: {list_var}")

# Adding elements
list_var.append(False)
print(f"List after append: {list_var}")
```

```
List: [1, 'apple', 3.14, True], Type: <class 'list'>
First element: 1
Slice (elements 1 and 2): ['apple', 3.14]
Modified list: [1, 'orange', 3.14, True]
List after append: [1, 'orange', 3.14, True, False]
```

Common List Methods

Python lists come with several built-in methods that allow you to modify, access, and manipulate their contents.

- **list.append(item)**: Adds a single item to the end of the list.
- **list.extend(iterable)**: Appends all the items from an iterable (like another list, tuple, or string) to the end of the list.
- **list.insert(index, item)**: Inserts an item at a specified index in the list.
- **list.remove(item)**: Removes the first occurrence of a specified item from the list. Raises a ValueError if the item is not found.

- **list.pop([index])**: Removes and returns the item at the given index. If no index is specified, pop() removes and returns the last item in the list.
- **list.clear()**: Removes all items from the list, making it empty.
- **list.index(item[, start[, end]])**: Returns the index of the first occurrence of the specified item. Optional start and end arguments can limit the search to a sub-sequence of the list. Raises a ValueError if the item is not found.
- **list.count(item)**: Returns the number of times a specified item appears in the list.
- **list.sort(key=None, reverse=False)**: Sorts the items of the list in place (modifies the original list). key specifies a function to be called on each list element prior to making comparisons, and reverse=True sorts in descending order.
- **list.reverse()**: Reverses the order of elements in the list in place.
- **list.copy()**: Returns a shallow copy of the list.

```
# Initial list
my_list = [10, 20, 30, 40, 50]
print(f"Original List: {my_list}")

# 1. append()
my_list.append(60)
print(f"After append(60): {my_list}")

# 2. extend()
another_list = [70, 80]
my_list.extend(another_list)
print(f"After extend([70, 80]): {my_list}")

# 3. insert()
my_list.insert(1, 15) # Insert 15 at index 1
print(f"After insert(1, 15): {my_list}")
```

```
# 4. remove()
my_list.remove(30) # Remove the first occurrence of 30
print(f"After remove(30): {my_list}")

# 5. pop()
popped_item = my_list.pop() # Removes and returns the last item
print(f"After pop() (removed {popped_item}): {my_list}")

popped_at_index = my_list.pop(0) # Removes and returns item at index 0
print(f"After pop(0) (removed {popped_at_index}): {my_list}")

# 6. index()
index_of_40 = my_list.index(40)
print(f"Index of 40: {index_of_40}")

# 7. count()
my_list.append(40) # Add another 40 for counting
count_of_40 = my_list.count(40)
print(f"Count of 40: {count_of_40}")
print(f"List before sort: {my_list}")

# 8. sort()
my_list.sort() # Sorts in ascending order in place
print(f"After sort(): {my_list}")

my_list.sort(reverse=True) # Sorts in descending order
```

```
print(f"After sort(reverse=True): {my_list}")

# 9. reverse()
my_list.reverse() # Reverses the order in place
print(f"After reverse(): {my_list}")

# 10. copy()
copy_list = my_list.copy()
print(f"Copied list: {copy_list}")

# Demonstrate independent modification
my_list.append(99)
print(f"Original list after append: {my_list}")
print(f"Copied list (unchanged): {copy_list}")

# 11. clear()
my_list.clear()
print(f"After clear(): {my_list}")
```

```
Original List: [10, 20, 30, 40, 50]
After append(60): [10, 20, 30, 40, 50, 60]
After extend([70, 80]): [10, 20, 30, 40, 50, 60, 70, 80]
After insert(1, 15): [10, 15, 20, 30, 40, 50, 60, 70, 80]
After remove(30): [10, 15, 20, 40, 50, 60, 70, 80]
After pop() (removed 80): [10, 15, 20, 40, 50, 60, 70]
After pop(0) (removed 10): [15, 20, 40, 50, 60, 70]
Index of 40: 2
```

```
Count of 40: 2

List before sort: [15, 20, 40, 50, 60, 70, 40]

After sort(): [15, 20, 40, 40, 50, 60, 70]

After sort(reverse=True): [70, 60, 50, 40, 40, 20, 15]

After reverse(): [15, 20, 40, 40, 50, 60, 70]

Copied list: [15, 20, 40, 40, 50, 60, 70]

Original list after append: [15, 20, 40, 40, 50, 60, 70, 99]

Copied list (unchanged): [15, 20, 40, 40, 50, 60, 70]

After clear(): []
```

5. Tuples (tuple)

Tuples are ordered, immutable sequences of items. Like lists, they can contain elements of different datatypes, but once a tuple is created, its contents cannot be changed. Tuples are defined using parentheses ().

Why use them?

- Used for collections of items that should not change (e.g., geographical coordinates, database records).
- Can be used as keys in dictionaries (unlike lists) because they are immutable.

```
tuple_var = (1, 'banana', 2.71, False)

print(f"Tuple: {tuple_var}, Type: {type(tuple_var)}")


# Accessing elements

print(f"Second element: {tuple_var[1]}")


# Trying to modify (will cause an error)

# tuple_var[1] = 'grape' # Uncommenting this line will raise a TypeError
```

Common Tuple Methods

Due to their immutability, tuples have fewer built-in methods compared to lists. The most commonly used methods are:

- **tuple.count(item)**: Returns the number of times a specified item appears in the tuple.
- **tuple.index(item[, start[, end]])**: Returns the index of the first occurrence of the specified item. Raises a ValueError if the item is not found.

```
# Initial tuple
my_tuple = (10, 20, 30, 20, 40, 50)
print(f"Original Tuple: {my_tuple}")

# 1. count()
count_of_20 = my_tuple.count(20)
print(f"Count of 20: {count_of_20}")

# 2. index()
index_of_30 = my_tuple.index(30)
print(f"Index of 30: {index_of_30}")

# Trying to find an item not in the tuple (will raise a ValueError)
# try:
#     my_tuple.index(99)
# except ValueError as e:
#     print(f"Error: {e}")
```

```
Original Tuple: (10, 20, 30, 20, 40, 50)
Count of 20: 2
Index of 30: 2
```

7. Sets (set)

Sets are unordered collections of unique elements. They are mutable. Sets are defined using curly braces {} or the set() constructor. Duplicate elements are automatically removed.

Why use them?

- Useful for storing unique items, performing mathematical set operations (union, intersection, difference), and quickly checking for membership.

```
set_var = {1, 2, 3, 2, 4}

print(f"Set: {set_var}, Type: {type(set_var)}") # Notice the duplicate '2' is removed


# Adding elements

set_var.add(5)

print(f"Set after adding 5: {set_var}")


# Removing elements

set_var.remove(1)

print(f"Set after removing 1: {set_var}")


# Set operations

set1 = {1, 2, 3}

set2 = {3, 4, 5}

print(f"Union: {set1.union(set2)}")

print(f"Intersection: {set1.intersection(set2)}")
```

```
Set: {1, 2, 3, 4}, Type: <class 'set'>

Set after adding 5: {1, 2, 3, 4, 5}

Set after removing 1: {2, 3, 4, 5}

Union: {1, 2, 3, 4, 5}

Intersection: {3}
```

Common Set Methods

Python sets offer various methods for adding, removing, and performing mathematical operations on their elements.

- **set.add(element)**: Adds a given element to the set. If the element is already present, it does nothing.
- **set.remove(element)**: Removes the specified element from the set. Raises a KeyError if the element is not found.
- **set.discard(element)**: Removes the specified element from the set if it is present. Does nothing if the element is not found (no error).
- **set.pop()**: Removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.
- **set.clear()**: Removes all elements from the set.
- **set.union(other_set) or set1 | set2**: Returns a new set containing all unique elements from both sets.
- **set.intersection(other_set) or set1 & set2**: Returns a new set containing only the elements common to both sets.
- **set.difference(other_set) or set1 - set2**: Returns a new set containing elements in the first set but not in the second.
- **set.symmetric_difference(other_set) or set1 ^ set2**: Returns a new set containing elements that are in either set, but not in both.
- **set.issubset(other_set)**: Returns True if all elements of the set are present in other_set.
- **set.issuperset(other_set)**: Returns True if all elements of other_set are present in the set.
- **set.isdisjoint(other_set)**: Returns True if the set has no elements in common with other_set.

```
# Initial set

my_set = {10, 20, 30, 40}

print(f"Original Set: {my_set}")
```

```
# 1. add()
my_set.add(50)
my_set.add(20) # Adding an existing element does nothing
print(f"After add(50): {my_set}")

# 2. remove()
try:
    my_set.remove(10)
    print(f"After remove(10): {my_set}")
except KeyError as e:
    print(f"Error removing: {e}")

# 3. discard()
my_set.discard(60) # Discarding a non-existent element does not raise an error
print(f"After discard(60): {my_set}")

# 4. pop()
popped_element = my_set.pop()
print(f"After pop() (removed {popped_element}): {my_set}")

# Set operations
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}

print(f"Set A: {set_a}")
print(f"Set B: {set_b}")
print(f"Union (A | B): {set_a | set_b}")
print(f"Intersection (A & B): {set_a & set_b}")
print(f"Difference (A - B): {set_a - set_b}")
print(f"Symmetric Difference (A ^ B): {set_a ^ set_b}")

set_c = {1, 2}
print(f"Is set_c a subset of set_a? {set_c.issubset(set_a)}")
print(f"Is set_a a superset of set_c? {set_a.issuperset(set_c)}")

set_d = {7, 8}
print(f"Are set_a and set_d disjoint? {set_a.isdisjoint(set_d)}")

# 5. clear()
my_set.clear()
print(f"After clear(): {my_set}")
```

```
Original Set: {40, 10, 20, 30}
After add(50): {40, 10, 50, 20, 30}
After remove(10): {40, 50, 20, 30}
After discard(60): {40, 50, 20, 30}
After pop() (removed 40): {50, 20, 30}
Set A: {1, 2, 3, 4}
Set B: {3, 4, 5, 6}
Union (A | B): {1, 2, 3, 4, 5, 6}
Intersection (A & B): {3, 4}
Difference (A - B): {1, 2}
Symmetric Difference (A ^ B): {1, 2, 5, 6}
```

```
Is set_c a subset of set_a? True

Is set_a a superset of set_c? True

Are set_a and set_d disjoint? True

After clear(): set()
```

6. Dictionaries (dict)

Dictionaries are unordered collections of key-value pairs. Each key must be unique and immutable, while values can be of any datatype and can be changed. Dictionaries are defined using curly braces {}.

Why use them?

- Excellent for storing structured data where you need to associate a value with a specific key (e.g., user profiles, configuration settings).

```
dict_var = {'name': 'Alice', 'age': 30, 'city': 'New York'}

print(f"Dictionary: {dict_var}, Type: {type(dict_var)}")


# Accessing values by key

print(f"Name: {dict_var['name']}")


# Modifying a value

dict_var['age'] = 31

print(f"Modified age: {dict_var['age']}")


# Adding a new key-value pair

dict_var['occupation'] = 'Engineer'

print(f"Dictionary after adding new key: {dict_var}")
```

```
Dictionary: {'name': 'Alice', 'age': 30, 'city': 'New York'}, Type: <class 'dict'>

Name: Alice
```

```
Modified age: 31

Dictionary after adding new key: {'name': 'Alice', 'age': 31, 'city': 'New York', 'occupation': 'Engineer'}
```

Common Dictionary Methods

Python dictionaries come with a rich set of methods for manipulating and accessing their key-value pairs.

- **dict.clear()**: Removes all items from the dictionary.
- **dict.copy()**: Returns a shallow copy of the dictionary.
- **dict.fromkeys(iterable, value=None)**: Returns a new dictionary with keys from iterable and values equal to value (defaulting to None).
- **dict.get(key, default=None)**: Returns the value for key if key is in the dictionary, else default. If default is not given, it defaults to None.
- **dict.items()**: Returns a new view of the dictionary's items (key, value) pairs.
- **dict.keys()**: Returns a new view of the dictionary's keys.
- **dict.pop(key, default)**: Removes key from the dictionary and returns its value. If key is not found, default is returned if given, otherwise KeyError is raised.
- **dict.popitem()**: Removes and returns a (key, value) pair from the dictionary. Pairs are returned in LIFO (Last In, First Out) order in Python 3.7+.
- **dict.setdefault(key, default=None)**: If key is in the dictionary, return its value. If not, insert key with a value of default and return default.
- **dict.update([other])**: Updates the dictionary with the key/value pairs from other, overwriting existing keys.
- **dict.values()**: Returns a new view of the dictionary's values.

```
# Initial dictionary

my_dict = {'name': 'Charlie', 'age': 25, 'city': 'London'}

print(f"Original Dictionary: {my_dict}")


# 1. get()
```

```
name = my_dict.get('name')
country = my_dict.get('country', 'Unknown') # With default value
print(f"Name (using get): {name}")
print(f"Country (using get with default): {country}")

# 2. keys(), values(), items()
print(f"Keys: {my_dict.keys()}")
print(f"Values: {my_dict.values()}")
print(f"Items: {my_dict.items()}")

# 3. update()
my_dict.update({'age': 26, 'occupation': 'Artist'})
print(f"After update: {my_dict}")

# 4. pop()
popped_occupation = my_dict.pop('occupation')
print(f"After pop('occupation') (removed {popped_occupation}): {my_dict}")

# 5. setdefault()
hobby = my_dict.setdefault('hobby', 'reading')
print(f"After setdefault('hobby', 'reading'): {my_dict}, returned: {hobby}")

# Trying to setdefault for existing key
age = my_dict.setdefault('age', 30) # Will return existing age, not set to 30
print(f"After setdefault('age', 30): {my_dict}, returned: {age}")

# 6. copy()
```

```
copy_dict = my_dict.copy()

print(f"Copied dictionary: {copy_dict}")


# Demonstrate independent modification

my_dict['new_key'] = 'new_value'

print(f"Original dictionary after modification: {my_dict}")

print(f"Copied dictionary (unchanged): {copy_dict}")


# 7. popitem() (removes an arbitrary item, typically last inserted in Python 3.7+)

popped_pair = my_dict.popitem()

print(f"After popitem() (removed {popped_pair}): {my_dict}")


# 8. clear()

my_dict.clear()

print(f"After clear(): {my_dict}")


# 9. fromkeys()

keys = ['a', 'b', 'c']

new_dict_from_keys = dict.fromkeys(keys, 0)

print(f"New dictionary fromkeys: {new_dict_from_keys}")
```

```
Original Dictionary: {'name': 'Charlie', 'age': 25, 'city': 'London'}

Name (using get): Charlie

Country (using get with default): Unknown

Keys: dict_keys(['name', 'age', 'city'])

Values: dict_values(['Charlie', 25, 'London'])

Items: dict_items([('name', 'Charlie'), ('age', 25), ('city', 'London')])
```

```
After update: {'name': 'Charlie', 'age': 26, 'city': 'London', 'occupation': 'Artist'}

After pop('occupation') (removed Artist): {'name': 'Charlie', 'age': 26, 'city': 'London'}

After setdefault('hobby', 'reading'): {'name': 'Charlie', 'age': 26, 'city': 'London', 'hobby':
'reading'}, returned: reading

After setdefault('age', 30): {'name': 'Charlie', 'age': 26, 'city': 'London', 'hobby': 'reading'},
returned: 26

Copied dictionary: {'name': 'Charlie', 'age': 26, 'city': 'London', 'hobby': 'reading'}

Original dictionary after modification: {'name': 'Charlie', 'age': 26, 'city': 'London', 'hobby':
'reading', 'new_key': 'new_value'}

Copied dictionary (unchanged): {'name': 'Charlie', 'age': 26, 'city': 'London', 'hobby': 'reading'}

After popitem() (removed ('new_key', 'new_value')): {'name': 'Charlie', 'age': 26, 'city': 'London',
'hobby': 'reading'}

After clear(): {}

New dictionary fromkeys: {'a': 0, 'b': 0, 'c': 0}
```

NumPy (Numerical Python)

Concept: NumPy is a fundamental library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays efficiently. The core of NumPy is the ndarray object.

Why use it?

- **Performance**: NumPy arrays are stored more compactly and are optimized for numerical operations, making them significantly faster than Python lists for large datasets.
- **Powerful N-dimensional arrays**: Allows for efficient storage and manipulation of data in grids (matrices), vectors, and higher dimensions.
- **Mathematical Functions**: Provides a vast array of mathematical functions for array operations, linear algebra, Fourier transforms, random number generation, etc.
- **Interoperability**: Forms the basis for many other scientific computing libraries in Python (e.g., SciPy, Pandas, Matplotlib).

```
import numpy as np
```

```
print("--- 1D Array Example ---")
# Create a NumPy array from a Python list
np_array_1d = np.array([1, 2, 3, 4, 5])
print(f"1D Array: {np_array_1d}")
print(f"Type: {type(np_array_1d)}")
print(f"Shape: {np_array_1d.shape}")
print(f"Data Type: {np_array_1d.dtype}")

print("\n")

print("--- 2D Array Example ---")
# Create a 2D NumPy array (matrix)
np_array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(f"2D Array:\n{np_array_2d}")
print(f"Type: {type(np_array_2d)}")
print(f"Shape: {np_array_2d.shape}")
print(f"Data Type: {np_array_2d.dtype}")

print("\n")

print("--- Element-wise Operations ---")
# Perform element-wise operations (much faster than lists)
array_a = np.array([10, 20, 30])
array_b = np.array([1, 2, 3])

sum_array = array_a + array_b
product_array = array_a * array_b
```

```
print(f"Array A: {array_a}")
print(f"Array B: {array_b}")
print(f"Sum (A + B): {sum_array}")
print(f"Product (A * B): {product_array}")

print("\n")

print("--- Universal Functions (ufuncs) ---")
# Universal functions (ufuncs) - apply functions element-wise
sqrt_array = np.sqrt(array_a)
print(f"Square root of A: {sqrt_array}")

print("\n")

print("--- Matrix Multiplication (Dot Product) ---")
# Dot product (matrix multiplication)
matrix_c = np.array([[1, 2], [3, 4]])
matrix_d = np.array([[5, 6], [7, 8]])

dot_product = np.dot(matrix_c, matrix_d)
print(f"Matrix C:\n{matrix_c}")
print(f"Matrix D:\n{matrix_d}")
print(f"Dot product (C . D):\n{dot_product}")
```

```
--- 1D Array Example ---
1D Array: [1 2 3 4 5]
```

```
Type: <class 'numpy.ndarray'>
Shape: (5,)
Data Type: int64

------------------------------

--- 2D Array Example ---
2D Array:
[[1 2 3]
 [4 5 6]]
Type: <class 'numpy.ndarray'>
Shape: (2, 3)
Data Type: int64

------------------------------

--- Element-wise Operations ---
Array A: [10 20 30]
Array B: [1 2 3]
Sum (A + B): [11 22 33]
Product (A * B): [10 40 90]
------------------------------
--- Universal Functions (ufuncs) ---
Square root of A: [3.16227766 4.47213595 5.47722558]
------------------------------
--- Matrix Multiplication (Dot Product) ---
Matrix C:
```

```
[[1 2]
 [3 4]]
Matrix D:
[[5 6]
 [7 8]]
Dot product (C . D):
[[19 22]
 [43 50]]
```

Pandas (Python Data Analysis Library)

Concept: Pandas is a powerful, open-source data manipulation and analysis library for Python. It provides data structures like DataFrame and Series that make working with tabular data (like spreadsheets or SQL tables) incredibly intuitive and efficient. It's built on top of NumPy, meaning it offers high-performance operations.

- **DataFrame**: A two-dimensional, size-mutable, tabular data structure with labeled axes (rows and columns). Think of it as a spreadsheet or a SQL table.
- **Series**: A one-dimensional labeled array capable of holding any data type. You can think of it as a single column of a DataFrame.

Why use it?

- **Easy Data Handling**: Simplifies loading, cleaning, transforming, and analyzing diverse datasets.
- **Robust Data Structures**: DataFrames and Series are optimized for tabular data operations.
- **Powerful Operations**: Offers extensive functionalities for data alignment, merging, reshaping, grouping, and more.
- **Missing Data Handling**: Provides tools for easily handling missing data (NaN values).
- **Time Series Functionality**: Excellent support for time-series data.
- **Integration**: Seamlessly integrates with other Python libraries like NumPy, Matplotlib, and Scikit-learn, making it a cornerstone of the data science ecosystem.

```
import pandas as pd


print("--- Creating a DataFrame ---")
# Create a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [24, 27, 22, 32, 29],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Miami'],
    'Salary': [70000, 85000, 60000, 95000, 78000]
}
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)


print("\n")


print("--- Accessing Columns (Series) ---")
# Access a single column (returns a Series)
names = df['Name']
print("Names (Series):")
print(names)
print(f"Type of names: {type(names)}")


print("\n")


print("--- Basic Data Selection (.loc and .iloc) ---")
# Select row by label (using .loc)
```

```
row_1_loc = df.loc[1]
print("Row at index 1 (using .loc):")
print(row_1_loc)

# Select row by integer position (using .iloc)
row_0_iloc = df.iloc[0]
print("\nRow at position 0 (using .iloc):")
print(row_0_iloc)

# Select specific cells
salary_bob = df.loc[1, 'Salary']
print(f"\nBob's Salary: {salary_bob}")

print("\n")

print("--- Filtering Data ---")
# Filter DataFrame for people older than 25
older_than_25 = df[df['Age'] > 25]
print("People older than 25:")
print(older_than_25)

print("\n")

print("--- Adding a New Column ---")
# Add a new column based on existing data
df['Bonus'] = df['Salary'] * 0.10
print("DataFrame with 'Bonus' column:")
print(df)


print("\n")


print("--- Basic Aggregation (groupby) ---")
# Example: Group by City and calculate average salary (though not very meaningful with this
small data)
df_grouped_city = df.groupby('City')['Salary'].mean().reset_index()
print("Average Salary by City:")
print(df_grouped_city)


print("\n")


print("--- Descriptive Statistics ---")
# Get quick descriptive statistics for numerical columns
print("Descriptive statistics for numerical columns:")
print(df.describe())
```

```
--- Creating a DataFrame ---
Original DataFrame:
      Name  Age         City  Salary
0    Alice   24     New York   70000
1      Bob   27  Los Angeles   85000
2  Charlie   22      Chicago   60000
3    David   32      Houston   95000
4      Eve   29        Miami   78000
------------------------------
```

```
--- Accessing Columns (Series) ---
Names (Series):
0      Alice
1        Bob
2    Charlie
3      David
4        Eve
Name: Name, dtype: object
Type of names: <class 'pandas.core.series.Series'>

------------------------------

--- Basic Data Selection (.loc and .iloc) ---
Row at index 1 (using .loc):
Name              Bob
Age                27
City      Los Angeles
Salary          85000
Name: 1, dtype: object

Row at position 0 (using .iloc):
Name         Alice
Age             24
City      New York
Salary       70000
Name: 0, dtype: object
```

```
Bob's Salary: 85000

------------------------------

--- Filtering Data ---
People older than 25:
    Name  Age         City  Salary
1    Bob   27  Los Angeles   85000
3  David   32      Houston   95000
4    Eve   29        Miami   78000

------------------------------

--- Adding a New Column ---
DataFrame with 'Bonus' column:
      Name  Age         City  Salary   Bonus
0    Alice   24     New York   70000  7000.0
1      Bob   27  Los Angeles   85000  8500.0
2  Charlie   22      Chicago   60000  6000.0
3    David   32      Houston   95000  9500.0
4      Eve   29        Miami   78000  7800.0

------------------------------

--- Basic Aggregation (groupby) ---
Average Salary by City:
```

```
         City   Salary
0      Chicago  60000.0
1      Houston  95000.0
2  Los Angeles  85000.0
3        Miami  78000.0
4     New York  70000.0

------------------------------
```

```
--- Descriptive Statistics ---
Descriptive statistics for numerical columns:
             Age        Salary        Bonus
count   5.000000      5.000000     5.000000
mean   26.800000  77600.000000  7760.000000
std     3.962323  13464.768843  1346.476884
min    22.000000  60000.000000  6000.000000
25%    24.000000  70000.000000  7000.000000
50%    27.000000  78000.000000  7800.000000
75%    29.000000  85000.000000  8500.000000
max    32.000000  95000.000000  9500.000000
```

Performing Data Analysis with Dummy Data

For this section, we'll create a new dummy dataset to showcase various data analysis techniques using Pandas. This will demonstrate how to inspect, clean, filter, and aggregate data, which are fundamental steps in any data analysis workflow.

```
import pandas as pd
import numpy as np
```

```
print("--- Creating Dummy Data ---")
# Create a dictionary for our dummy data
dummy_data = {
    'ProductID': [f'P{i:03d}' for i in range(1, 11)],
    'Category': ['Electronics', 'Clothes', 'Electronics', 'Books', 'Clothes', 'Electronics', 'Books',
'Clothes', 'Electronics', 'Books'],
    'Price': np.random.randint(50, 500, size=10).astype(float), # Ensure dtype is float to allow
np.nan
    'QuantitySold': np.random.randint(1, 20, size=10).astype(float), # Ensure dtype is float to
allow np.nan
    'Region': ['East', 'West', 'North', 'South', 'East', 'West', 'North', 'South', 'East', 'West'],
    'Rating': np.random.uniform(2.5, 5.0, size=10).round(1)
}

# Introduce some missing values intentionally
dummy_data['Price'][2] = np.nan # Missing price
dummy_data['QuantitySold'][7] = np.nan # Missing quantity

# Create the DataFrame
df_dummy = pd.DataFrame(dummy_data)

print("Dummy DataFrame Created:")
print(df_dummy)
print("\n")

print("**Why this code?**\n")
print("- `import pandas as pd` and `import numpy as np`: Imports the necessary libraries. Pandas
is for DataFrame operations, and NumPy is used for creating numerical arrays and `np.nan` for
missing values.")
```

print("- `dummy_data = {...}`: A dictionary is created to define the column names and their corresponding data. We use list comprehensions (`f'P{i:03d}' for i in range(1, 11)`) and NumPy functions (`np.random.randint`, `np.random.uniform`) to generate varied data quickly.")

print("- `np.nan`: Manually inserting `np.nan` (Not a Number) values demonstrates how to simulate missing data, which is common in real datasets and needs to be handled.")

print("- `df_dummy = pd.DataFrame(dummy_data)`: This line converts the dictionary into a Pandas DataFrame, making it a structured table that we can easily analyze.")

print("- `print(df_dummy)`: Displays the entire DataFrame to review its contents.")

```
--- Creating Dummy Data ---
Dummy DataFrame Created:
  ProductID     Category  Price  QuantitySold Region  Rating
0      P001  Electronics  116.0          13.0   East     3.8
1      P002      Clothes  212.0           9.0   West     3.3
2      P003  Electronics    NaN           6.0  North     4.4
3      P004        Books  231.0          15.0  South     3.4
4      P005      Clothes  484.0           9.0   East     3.0
5      P006  Electronics  127.0          19.0   West     3.0
6      P007        Books  108.0          16.0  North     2.7
7      P008      Clothes  407.0           NaN  South     4.4
8      P009  Electronics  471.0          18.0   East     3.4
9      P010        Books  254.0          12.0   West     3.0
```

Why this code?

- `import pandas as pd` and `import numpy as np`: Imports the necessary libraries. Pandas is for DataFrame operations, and NumPy is used for creating numerical arrays and `np.nan` for missing values.

- `dummy_data = {...}`: A dictionary is created to define the column names and their corresponding data. We use list comprehensions (`f'P{i:03d}' for i in range(1, 11)`) and NumPy functions (`np.random.randint`, `np.random.uniform`) to generate varied data quickly.

- `np.nan`: Manually inserting `np.nan` (Not a Number) values demonstrates how to simulate missing data, which is common in real datasets and needs to be handled.

- `df_dummy = pd.DataFrame(dummy_data)`: This line converts the dictionary into a Pandas DataFrame, making it a structured table that we can easily analyze.

- `print(df_dummy)`: Displays the entire DataFrame to review its contents.

```
print("--- Data Inspection ---")
print("DataFrame Information (df.info()):")
df_dummy.info()

print("\n")

print("Descriptive Statistics (df.describe()):")
print(df_dummy.describe())

print("\n")

print("Missing Values (df.isnull().sum()):")
print(df_dummy.isnull().sum())

print("\n")

print("**Why this code?**\n")
print("- `df_dummy.info()`: Provides a concise summary of the DataFrame, including the data types of each column, the number of non-null values, and memory usage. This is crucial for initial data type checks and identifying columns with missing data.")
print("- `df_dummy.describe()`: Generates descriptive statistics for numerical columns, such as count, mean, standard deviation, min, max, and quartiles. This helps in understanding the distribution and central tendency of the numerical data.")

print("- `df_dummy.isnull().sum()`: Calculates the sum of null values for each column. This is a quick way to see how much missing data each column has, which is essential for deciding on a strategy to handle them.")
```

```
--- Data Inspection ---

DataFrame Information (df.info()):

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 10 entries, 0 to 9

Data columns (total 6 columns):

 #   Column        Non-Null Count  Dtype

---  ------        --------------  -----

 0   ProductID     10 non-null     object

 1   Category      10 non-null     object

 2   Price         9 non-null      float64

 3   QuantitySold  9 non-null      float64

 4   Region        10 non-null     object

 5   Rating        10 non-null     float64

dtypes: float64(3), object(3)

memory usage: 612.0+ bytes



Descriptive Statistics (df.describe()):

            Price  QuantitySold     Rating

count    9.000000       9.00000  10.000000

mean   267.777778      13.00000   3.440000
```

```
std    150.196353       4.41588   0.589161
min    108.000000       6.00000   2.700000
25%    127.000000       9.00000   3.000000
50%    231.000000      13.00000   3.350000
75%    407.000000      16.00000   3.700000
max    484.000000      19.00000   4.400000
```

```
Missing Values (df.isnull().sum()):
ProductID       0
Category        0
Price           1
QuantitySold    1
Region          0
Rating          0
dtype: int64
```

Why this code?

- `df_dummy.info()`: Provides a concise summary of the DataFrame, including the data types of each column, the number of non-null values, and memory usage. This is crucial for initial data type checks and identifying columns with missing data.

- `df_dummy.describe()`: Generates descriptive statistics for numerical columns, such as count, mean, standard deviation, min, max, and quartiles. This helps in understanding the distribution and central tendency of the numerical data.

- `df_dummy.isnull().sum()`: Calculates the sum of null values for each column. This is a quick way to see how much missing data each column has, which is essential for deciding on a strategy to handle them.

```
print("--- Handling Missing Data ---")
# Option 1: Fill missing numerical values with the mean
# For 'Price' and 'QuantitySold', we'll fill NaN with their respective means.
# We calculate the mean *before* filling to avoid including the NaN itself.
df_dummy['Price'] = df_dummy['Price'].fillna(df_dummy['Price'].mean())
df_dummy['QuantitySold'] =
df_dummy['QuantitySold'].fillna(df_dummy['QuantitySold'].mean())

print("DataFrame after filling missing numerical values with mean:")
print(df_dummy)

print("\n")

print("Missing Values after handling:")
print(df_dummy.isnull().sum())

print("\n")

print("**Why this code?**\n")
print("- `df_dummy['ColumnName'].fillna(value, inplace=True)`: This method is used to replace
missing values (NaN) in a specific column. We've chosen to fill numerical missing values
(`Price`, `QuantitySold`) with the mean of their respective columns. This is a common
imputation strategy that maintains the overall mean of the column and is suitable when the
amount of missing data is not too large. `inplace=True` modifies the DataFrame directly.")
print("- `df_dummy['ColumnName'].mean()`: Calculates the mean of the specified column,
which is then used as the `value` to fill the NaN entries.")
print("- Displaying the DataFrame and re-checking `isnull().sum()`: This confirms that the
missing values have been successfully handled and that the DataFrame is now 'cleaner' for
further analysis.")
```

```
--- Handling Missing Data ---
DataFrame after filling missing numerical values with mean:
  ProductID     Category       Price  QuantitySold Region  Rating
0      P001  Electronics  116.000000          13.0   East     3.8
1      P002      Clothes  212.000000           9.0   West     3.3
2      P003  Electronics  267.777778           6.0  North     4.4
3      P004        Books  231.000000          15.0  South     3.4
4      P005      Clothes  484.000000           9.0   East     3.0
5      P006  Electronics  127.000000          19.0   West     3.0
6      P007        Books  108.000000          16.0  North     2.7
7      P008      Clothes  407.000000          13.0  South     4.4
8      P009  Electronics  471.000000          18.0   East     3.4
9      P010        Books  254.000000          12.0   West     3.0


Missing Values after handling:
ProductID       0
Category        0
Price           0
QuantitySold    0
Region          0
Rating          0
dtype: int64
```

Why this code?

- `df_dummy['ColumnName'].fillna(value, inplace=True)`: This method is used to replace missing values (NaN) in a specific column. We've chosen to fill numerical missing values (`Price`, `QuantitySold`) with the mean of their respective columns. This is a common imputation strategy that maintains the overall mean of the column and is suitable when the amount of missing data is not too large. `inplace=True` modifies the DataFrame directly.

- `df_dummy['ColumnName'].mean()`: Calculates the mean of the specified column, which is then used as the `value` to fill the NaN entries.

- Displaying the DataFrame and re-checking `isnull().sum()`: This confirms that the missing values have been successfully handled and that the DataFrame is now 'cleaner' for further analysis.

```
print("--- Data Transformation and Feature Engineering ---")

# Create a new feature: 'TotalSales' = Price * QuantitySold

df_dummy['TotalSales'] = df_dummy['Price'] * df_dummy['QuantitySold']


# Create a categorical feature based on 'Rating'

df_dummy['RatingCategory'] = pd.cut(df_dummy['Rating'], bins=[0, 3.5, 4.5, 5.0],
labels=['Poor', 'Good', 'Excellent'])


print("DataFrame after adding 'TotalSales' and 'RatingCategory':")

print(df_dummy)


print("\n")


print("**Why this code?**\n")

print("- `df_dummy['TotalSales'] = df_dummy['Price'] * df_dummy['QuantitySold']`: This
creates a new, derived feature by combining existing columns. 'TotalSales' is a meaningful metric
often used in business analysis, showing the revenue generated by each product. This is a basic
form of feature engineering.")
```

```
print("- `pd.cut(df_dummy['Rating'], bins=[...], labels=[...])`: This function is used to segment
and sort data values into bins. Here, we're converting a continuous numerical feature (`Rating`)
into an ordinal categorical feature (`RatingCategory`). This can simplify analysis, especially
when visualizing or grouping data based on ranges rather than exact values.")
```

```
--- Data Transformation and Feature Engineering ---
DataFrame after adding 'TotalSales' and 'RatingCategory':
  ProductID     Category       Price  QuantitySold Region  Rating  \
0      P001  Electronics  116.000000          13.0   East     3.8
1      P002      Clothes  212.000000           9.0   West     3.3
2      P003  Electronics  267.777778           6.0  North     4.4
3      P004        Books  231.000000          15.0  South     3.4
4      P005      Clothes  484.000000           9.0   East     3.0
5      P006  Electronics  127.000000          19.0   West     3.0
6      P007        Books  108.000000          16.0  North     2.7
7      P008      Clothes  407.000000          13.0  South     4.4
8      P009  Electronics  471.000000          18.0   East     3.4
9      P010        Books  254.000000          12.0   West     3.0

    TotalSales RatingCategory
0  1508.000000           Good
1  1908.000000           Poor
2  1606.666667           Good
3  3465.000000           Poor
4  4356.000000           Poor
5  2413.000000           Poor
6  1728.000000           Poor
7  5291.000000           Good
```

```
8  8478.000000          Poor
9  3048.000000          Poor
```

Why this code?

- `df_dummy['TotalSales'] = df_dummy['Price'] * df_dummy['QuantitySold']`: This creates a new, derived feature by combining existing columns. 'TotalSales' is a meaningful metric often used in business analysis, showing the revenue generated by each product. This is a basic form of feature engineering.
- `pd.cut(df_dummy['Rating'], bins=[...], labels=[...])`: This function is used to segment and sort data values into bins. Here, we're converting a continuous numerical feature (`Rating`) into an ordinal categorical feature (`RatingCategory`). This can simplify analysis, especially when visualizing or grouping data based on ranges rather than exact values.

```
print("--- Data Filtering and Selection ---")
# Filter for products in 'Electronics' category with 'TotalSales' > 1000
electronics_high_sales = df_dummy[
    (df_dummy['Category'] == 'Electronics') &
    (df_dummy['TotalSales'] > 1000)
]

print("Electronics products with TotalSales > 1000:")
print(electronics_high_sales)

print("\n")

print("**Why this code?**\n")
print("- `df_dummy[...]`: This is how you select data from a DataFrame based on conditions. The `[...]` contains boolean conditions that return `True` for rows that match the criteria and `False` otherwise.")
print("- `(df_dummy['Category'] == 'Electronics')`: This is the first condition, selecting rows where the 'Category' column is 'Electronics'.")
print("- `(df_dummy['TotalSales'] > 1000)`: This is the second condition, selecting rows where 'TotalSales' is greater than 1000.")
print("- `&`: The ampersand symbol (`&`) acts as a logical 'AND' operator in Pandas, combining multiple conditions. Only rows that satisfy *both* conditions will be included in the `electronics_high_sales` DataFrame. This technique is fundamental for drilling down into specific subsets of your data.")
```

```
--- Data Filtering and Selection ---
Electronics products with TotalSales > 1000:
  ProductID     Category       Price  QuantitySold Region  Rating  \
0      P001  Electronics  116.000000          13.0   East     3.8
2      P003  Electronics  267.777778           6.0  North     4.4
5      P006  Electronics  127.000000          19.0   West     3.0
8      P009  Electronics  471.000000          18.0   East     3.4

    TotalSales RatingCategory
0  1508.000000           Good
2  1606.666667           Good
5  2413.000000           Poor
8  8478.000000           Poor
```

Why this code?

- `df_dummy[...]`: This is how you select data from a DataFrame based on conditions. The `[...]` contains boolean conditions that return `True` for rows that match the criteria and `False` otherwise.

- `(df_dummy['Category'] == 'Electronics')`: This is the first condition, selecting rows where the 'Category' column is 'Electronics'.

- `(df_dummy['TotalSales'] > 1000)`: This is the second condition, selecting rows where 'TotalSales' is greater than 1000.

- `&`: The ampersand symbol (`&`) acts as a logical 'AND' operator in Pandas, combining multiple conditions. Only rows that satisfy *both* conditions will be included in the `electronics_high_sales` DataFrame. This technique is fundamental for drilling down into specific subsets of your data.

```
print("--- Data Aggregation (Groupby) ---")
# Group by 'Category' and calculate the sum of 'TotalSales' and mean 'Rating'
category_summary = df_dummy.groupby('Category').agg(
    TotalRevenue=('TotalSales', 'sum'),
    AverageRating=('Rating', 'mean'),
    ProductCount=('ProductID', 'count')
).reset_index()

print("Summary by Category:")
print(category_summary)

print("\n")

# Group by 'Region' and find the average 'QuantitySold'
region_sales_avg = df_dummy.groupby('Region')['QuantitySold'].mean().reset_index()

print("Average Quantity Sold by Region:")
print(region_sales_avg)


print("\n")


print("**Why this code?**\n")
print("- `df_dummy.groupby('Category')`: This operation groups the DataFrame by unique values in the 'Category' column. Subsequent operations will be applied independently to each group.")
print("- `.agg(...)`: After grouping, `agg()` (aggregate) is used to perform multiple calculations on the groups. We define new column names (`TotalRevenue`, `AverageRating`, `ProductCount`) and specify the column to operate on and the aggregation function (`'sum'`, `'mean'`, `'count'`). This allows for powerful summary statistics.")
print("- `.reset_index()`: After `groupby()` and `agg()`, the grouping column ('Category' or 'Region') becomes the index. `reset_index()` converts it back into a regular column, making the output easier to work with.")
print("- This aggregation helps in understanding performance across different categories or regions, identifying top-performing areas, or spotting potential issues.")
```

```
--- Data Aggregation (Groupby) ---
Summary by Category:
      Category  TotalRevenue  AverageRating  ProductCount
0        Books   8241.000000       3.033333             3
1      Clothes  11555.000000       3.566667             3
2  Electronics  14005.666667       3.650000             4



Average Quantity Sold by Region:
  Region  QuantitySold
0   East     13.333333
1  North     11.000000
```

```
2  South     14.000000
3   West     13.333333
```

Why this code?

- `df_dummy.groupby('Category')`: This operation groups the DataFrame by unique values in the 'Category' column. Subsequent operations will be applied independently to each group.

- `.agg(...)`: After grouping, `agg()` (aggregate) is used to perform multiple calculations on the groups. We define new column names (`TotalRevenue`, `AverageRating`, `ProductCount`) and specify the column to operate on and the aggregation function (`'sum'`, `'mean'`, `'count'`). This allows for powerful summary statistics.

- `.reset_index()`: After `groupby()` and `agg()`, the grouping column ('Category' or 'Region') becomes the index. `reset_index()` converts it back into a regular column, making the output easier to work with.

- This aggregation helps in understanding performance across different categories or regions, identifying top-performing areas, or spotting potential issues.