

MODULE 5: Object oriented programming: Objects are mutable, Sameness, Copying.

MUTABILITY

In Python, every value—whether a number, string, list, or object—is represented internally as an **object** stored in memory. These objects have two main aspects: **identity** (their memory address) and **state** (the data they hold).

When we say an object is **mutable**, we mean that its **state** can be changed after it is created, while its **identity** remains the same.

For example, if we create a rectangle object, we can change its width, height, or position without creating a new rectangle. This ability to modify an object's internal attributes without replacing it entirely is what defines mutability.

Let us consider a class `Rectangle` that has a corner point, a width, and a height. The rectangle can be changed in size or moved to another position.

```
class Rectangle:
```

```
    def __init__(self, corner, width, height):
```

```
        self.corner = corner
```

```
        self.width = width
```

```
        self.height = height
```

```
    def grow(self, delta_width, delta_height):
```

```
        self.width += delta_width
```

```
        self.height += delta_height
```

```
    def move(self, dx, dy):
```

```
        self.corner.x += dx
```

```
self.corner.y += dy
```

Here, `grow()` and `move()` are methods that modify the object's state.

If we create a rectangle:

```
r = Rectangle(Point(10, 5), 100, 50)
```

```
r.grow(25, -10)
```

```
r.move(-10, 10)
```

After these operations, the **same rectangle object** has changed. Its identity in memory is still the same, but the internal values of its attributes (width, height, and corner) have been altered. This illustrates mutability clearly — the object persists, but its contents evolve.

Mutability makes Python powerful and flexible, especially when dealing with dynamic data structures like lists, dictionaries, and class instances. However, it also introduces complexity because multiple references may point to the same object, leading to unexpected side effects.

SAMENESS (IDENTITY) AND EQUALITY

The concept of *sameness* in Python can be tricky because the word “same” can mean two different things. When we say two objects are “the same,” do we mean they occupy the **same space in memory** or that they **contain the same data**?

This distinction mirrors real-life examples. If we say “Alice and Bob have the same car,” we might mean both have cars of the same model and color — two different cars with similar characteristics. But if we say “Alice and Bob have the same mother,” we mean there is only one person — the very same mother. In Python, the first case is **equality of contents**, while the second is **identity**.

Python expresses identity using the `is` operator. It tells us whether two names refer to the **same object in memory**.

For example:

```
p1 = Point(3, 4)
```

```
p2 = Point(3, 4)
```

```
print(p1 is p2) # False
```

Even though both points have identical coordinates, they are stored in two separate places in memory. They are equal in value but not identical in identity.

If we instead assign one variable to another:

```
p3 = p1
```

```
print(p1 is p3) # True
```

Now, both names point to the very same object in memory. Any change made through one variable will reflect through the other because both refer to the same entity.

This concept is often called **shallow equality** or **identity equality**, since it only checks if two references point to the same object, not whether the objects hold the same data.

DEEP EQUALITY (COMPARING CONTENTS)

Sometimes we want to know whether two objects represent the same data rather than being literally the same object. This is called **deep equality**.

We can check this manually by comparing individual attributes. For example:

```
def same_coordinates(p1, p2):  
  
    return (p1.x == p2.x) and (p1.y == p2.y)
```

Now if we create two different points with the same coordinates:

```
p1 = Point(3, 4)
```

```
p2 = Point(3, 4)
```

```
print(same_coordinates(p1, p2)) # True
```

Even though the two points are distinct in memory, they contain equal data. This demonstrates deep equality — two different objects that represent the same information.

In many programming situations, **deep equality is what we actually care about**, since we're usually interested in whether two objects carry equivalent values, not whether they occupy the same physical space in memory.

The Meaning of == in Python

Python's == operator is designed to check equality, but its behavior depends on how the object's class defines it. For built-in types like integers, strings, and lists, == is implemented to compare the **contents** of the objects. For user-defined classes, however, == performs the same check as is by default, comparing object identities.

To see this difference, consider the following example:

```
p = Point(4, 2)
```

```
s = Point(4, 2)
```

```
print("== on Points returns", p == s)
```

```
a = [2, 3]
```

```
b = [2, 3]
```

```
print("== on lists returns", a == b)
```

The first comparison prints False because Point does not define a special method for equality; hence, Python checks identity.

The second comparison prints True because lists implement a built-in equality method that compares their contents element by element.

Python allows developers to customize this behavior by defining a method called `__eq__()` inside a class. This method determines what equality (`==`) means for that class. For instance, if we modify the Point class as follows:

```
class Point:
```

```
    def __init__(self, x, y):  
  
        self.x = x  
  
        self.y = y  
  
    def __eq__(self, other):  
  
        return self.x == other.x and self.y == other.y
```

Now, comparing two different Point objects with the same coordinates will return True. In this way, equality becomes a matter of logical equivalence in content rather than physical identity.

ALIASING

Aliasing occurs when two or more variables refer to the same object in memory. This happens automatically whenever we use simple assignment between objects.

For example:

```
p1 = Point(3, 4)  
  
p2 = p1
```

Here, no new object is created. Both p1 and p2 refer to the same Point instance. Changing one will affect the other:

```
p2.x = 10  
  
print(p1.x) # Output: 10
```

Because they are two names for the same object, Python doesn't duplicate the object — it simply creates a new reference to it. This can be convenient, but it can also cause subtle bugs if you accidentally modify shared data.

Aliasing is a direct consequence of Python's **reference model**, where variables do not hold the actual data but references (pointers) to objects stored elsewhere in memory. Every time you pass a mutable object to a function, that function receives a reference to the original object — not a copy. Thus, if the function modifies the object, those changes will be visible outside the function as well.

Copying Objects

To avoid aliasing problems, we often want to create a **new object** that is a duplicate of an existing one. Python provides the `copy` module for this purpose, offering two types of copying: **shallow copy** and **deep copy**.

Shallow Copy

A shallow copy creates a new outer object but does not recursively copy the objects that it refers to. Instead, it simply copies references to those inner objects.

For instance:

```
import copy
```

```
p1 = Point(3, 4)
```

```
p2 = copy.copy(p1)
```

In this case, `p1` and `p2` are distinct `Point` objects with the same values. Changing one will not affect the other because there are no nested objects inside. However, when the object being copied contains references to other objects, as in the case of a rectangle that contains a point, a shallow copy will still share those inner references.

Consider this example:

```
b1 = Rectangle(Point(10, 5), 100, 50)
```

```
b2 = copy.copy(b1)
```

Here, b1 and b2 are two distinct rectangles, but both refer to the same Point object as their corner. If we move one rectangle:

```
b1.move(5, 0)
```

the change will also appear in b2, because both share the same corner point. Shallow copies can thus lead to aliasing at a deeper level.

Deep Copy

A **deep copy** overcomes this problem by recursively copying every object that is contained inside another. That means not only the outer object but also all embedded objects are duplicated, resulting in a completely independent copy.

Continuing the previous example:

```
b2 = copy.deepcopy(b1)
```

Now b1 and b2 are entirely separate rectangles, and their corner points are also different objects. Modifying one rectangle or its corner will not affect the other.

Deep copying is particularly important for complex or nested objects, such as lists of lists, dictionaries containing other mutable objects, or class instances that contain references to other instances. It ensures that every level of the object structure is fully replicated, so there is no shared data anywhere between the copies.

Why Aliasing and Copying Matter

Understanding how Python handles object identity and copying is critical for writing reliable and predictable programs. When you use mutable objects, you must always be aware of whether you are working with a new copy of an object or an alias of the same one. Accidental aliasing can cause functions to modify data unexpectedly, while incorrect copying can result in redundant memory usage or inconsistent states.

For example, if you are working with graphical objects, like points, rectangles, or circles, and you want to change the shape of one without affecting another, you must ensure that you use deep copies rather than simple assignment or shallow copies.

At the same time, mutability allows us to design efficient programs that modify data in place without constantly recreating objects, which would be costly in terms of performance. The key is understanding when you need independent objects and when shared references are acceptable.



INHERITANCE: PURE FUNCTIONS ,MODIFIERS, GENERALIZATION, OPERATOR OVERLOADING, POLYMORPHISM.

INHERITANCE

Inheritance is an OOP concept that allows one class to acquire the properties (attributes) and behaviors (methods) of another class. It lets you **reuse existing code** and **build more specialized classes** without rewriting code.

The class that is inherited from is called the **parent** or **base class**, and the class that inherits is called the **child** or **derived class**.

Why Use Inheritance?

Inheritance is useful because it:

- Promotes **code reuse**: Shared functionality can reside in the base class.
- Helps in **organizing code**: Common attributes and methods are centralized.
- Allows **specialization**: Child classes can add or override behavior without affecting the base class.
- Supports **polymorphism**: Functions can work with objects of different types that share the same base class.

Basic Syntax

In Python, a class inherits from another class by specifying the parent class in parentheses after the class name:

```
class Parent:
```

```
    def greet(self):
```

```
        print("Hello from Parent")
```

```
class Child(Parent):  
  
    pass # Inherits everything from Parent  
  
c = Child()  
  
c.greet() # Output: Hello from Parent
```

In this example, Child inherits the greet method from Parent without writing it again.

OVERRIDING METHODS

A child class can **override** a method of the parent class by defining a method with the same name. The child's version will take precedence.

```
class Parent:  
  
    def greet(self):  
  
        print("Hello from Parent")  
  
class Child(Parent):  
  
    def greet(self):  
  
        print("Hello from Child")  
  
c = Child()  
  
c.greet() # Output: Hello from Child
```

You can still call the parent's method explicitly using super():

```
class Child(Parent):  
  
    def greet(self):  
  
        super().greet() # Calls Parent's greet
```

```
print("Hello from Child")
```

```
c = Child()
```

```
c.greet()
```

Output:

```
# Hello from Parent
```

```
# Hello from Child
```

Adding Attributes in Child Classes

Child classes can also define new attributes in addition to those inherited:

```
class Person:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
class Employee(Person):
```

```
    def __init__(self, name, emp_id):
```

```
        super().__init__(name) # Initialize base class
```

```
        self.emp_id = emp_id
```

```
e = Employee("Alice", 101)
```

```
print(e.name) # Output: Alice
```

```
print(e.emp_id) # Output: 101
```

Here, Employee inherits the name attribute from Person but adds its own emp_id attribute.

Types of Inheritance

Python supports several types of inheritance:

1. **Single Inheritance:** A child inherits from a single parent.
2. **Multiple Inheritance:** A child inherits from multiple parents.

```
class Parent1:

    def greet1(self):

        print("Hello from Parent1")

class Parent2:

    def greet2(self):

        print("Hello from Parent2")

class Child(Parent1, Parent2):

    pass

c = Child()

c.greet1() # Hello from Parent1

c.greet2() # Hello from Parent2
```

3. **Multilevel Inheritance:** A class inherits from a child, forming a chain.

```
class Grandparent:

    def greet(self):

        print("Hello from Grandparent")

class Parent(Grandparent):
```

```
pass

class Child(Parent):

    pass

c = Child()

c.greet() # Hello from Grandparent
```

4. **Hierarchical Inheritance:** Multiple children inherit from the same parent.
5. **Hybrid Inheritance:** Combination of multiple types above.

The super() Function

super() is used in child classes to **call methods of the parent class**, especially in constructors (`__init__`) to ensure the base class is initialized properly. This is crucial when the base class has important setup code.

Advantages of Inheritance

- Reduces **code duplication**
- Makes programs more **organized** and **maintainable**
- Supports **polymorphism**, making functions more flexible
- Facilitates **extension** of code with minimal changes

Example: Vehicles

```
class Vehicle:

    def __init__(self, brand, wheels):

        self.brand = brand
```

```

        self.wheels = wheels

    def info(self):

        print(f'{self.brand} has {self.wheels} wheels')

class Car(Vehicle):

    def __init__(self, brand, wheels, doors):

        super().__init__(brand, wheels)

        self.doors = doors

    def info(self):

        super().info()

        print(f'It has {self.doors} doors')

my_car = Car("Toyota", 4, 4)

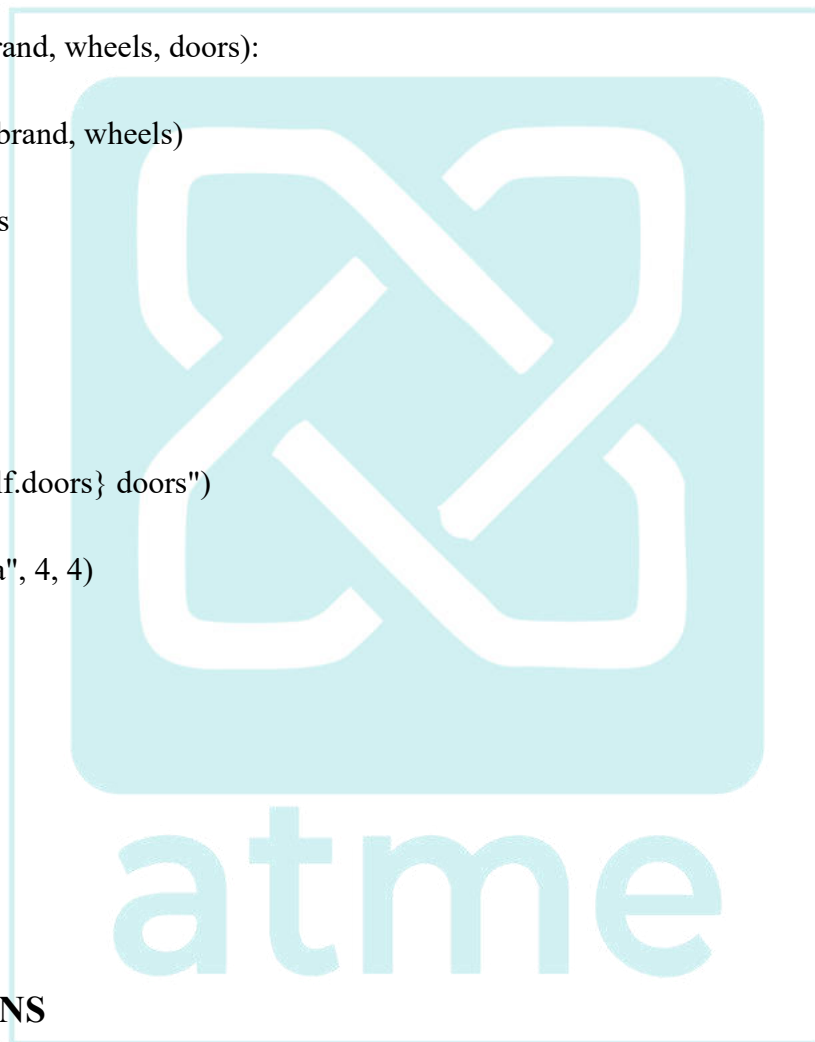
my_car.info()

# Output:

# Toyota has 4 wheels

# It has 4 doors

```



PURE FUNCTIONS

A pure function is one that does not modify its arguments or produce side effects such as printing or reading input. It simply takes input values, performs computations, and returns a result. In object-oriented programming, this often means that the function creates a new object rather than altering an existing one.

For example, when adding two time objects together, we can create a new MyTime object that represents the sum of the two times. This function does not modify the original time objects.

```
def add_time(t1, t2):
```

```
    h = t1.hours + t2.hours
```

```
    m = t1.minutes + t2.minutes
```

```
    s = t1.seconds + t2.seconds
```

```
    sum_t = MyTime(h, m, s)
```

```
    return sum_t
```

Here, the function `add_time` combines two `MyTime` objects, `t1` and `t2`, but does not alter either of them. Instead, it creates and returns a new time object. This is what makes it a pure function.

However, this simple approach has a limitation. If the sum of seconds or minutes exceeds 59, it will not automatically carry over to the next unit. To fix this, we can normalize the time.

```
def add_time(t1, t2):
```

```
    h = t1.hours + t2.hours
```

```
    m = t1.minutes + t2.minutes
```

```
    s = t1.seconds + t2.seconds
```

```
    if s >= 60:
```

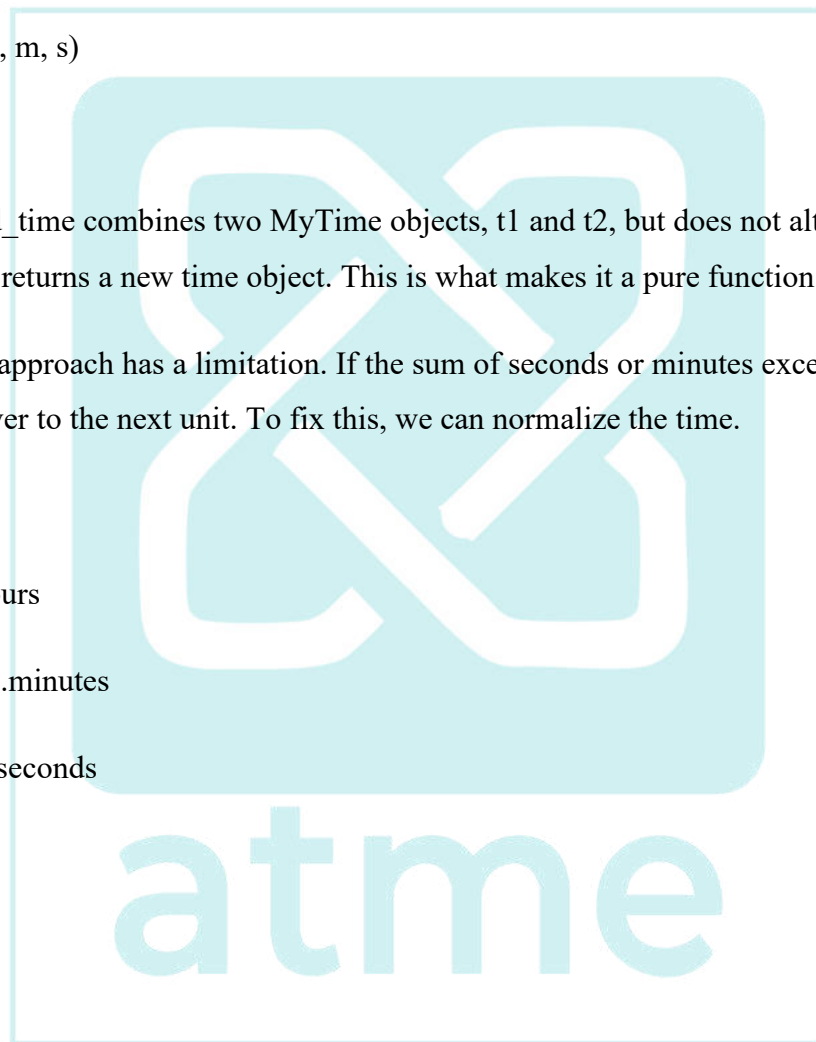
```
        s -= 60
```

```
        m += 1
```

```
    if m >= 60:
```

```
        m -= 60
```

```
        h += 1
```



```
return MyTime(h, m, s)
```

This function still remains pure because it creates and returns a new object, leaving the original time values unchanged.

Pure functions are predictable, testable, and easy to reuse since they depend only on their inputs and produce consistent outputs without side effects.

MODIFIERS

A modifier function changes the state of the object it receives as a parameter. It alters the object directly instead of returning a new one. This kind of function is known as a mutator because it mutates or modifies the object's attributes.

For example, consider the following function which adds seconds to a MyTime object:

```
def increment(t, secs):  
  
    t.seconds += secs  
  
    while t.seconds >= 60:  
  
        t.seconds -= 60  
  
        t.minutes += 1  
  
    while t.minutes >= 60:  
  
        t.minutes -= 60  
  
        t.hours += 1
```

This function changes the values stored inside the object `t`. After the function call, the object itself is different. This is the opposite of a pure function because it modifies its argument.

While modifiers can be more efficient in terms of memory and performance, they can also make debugging harder. This is because changes in one place can have unexpected effects in another if multiple references

point to the same object.

Converting a Function into a Method

Instead of writing functions that take objects as arguments, we can define the behavior inside the class itself using methods. This approach keeps the behavior close to the data and makes the program easier to understand.

For instance, instead of having a standalone function `increment(t, secs)`, we can make it a method inside the `MyTime` class:

```
class MyTime:

    def increment(self, seconds):

        self.seconds += seconds

        while self.seconds >= 60:

            self.seconds -= 60

            self.minutes += 1

        while self.minutes >= 60:

            self.minutes -= 60

            self.hours += 1
```

Now, the method can be called directly on an object:

```
current_time.increment(500)
```

Here, the variable `current_time` becomes the object referred to by `self` within the method. The method modifies the object in place.

This is an example of encapsulation, which means combining data and the methods that operate on that data

within a single class definition. It helps maintain clarity and control over how data is accessed and modified.

Treating Time as a Base-60 System

One useful insight in programming with time objects is to recognize that time values work in a base-60 system. One minute equals 60 seconds, and one hour equals 60 minutes. Thus, a MyTime object's data can be treated like digits in base 60.

Instead of manually handling the carry-over of seconds and minutes, we can convert all the time units into a single number of seconds, perform arithmetic, and then convert back.

For example:

```
def to_seconds(self):  
    return self.hours * 3600 + self.minutes * 60 + self.seconds
```

With this conversion, adding two time objects becomes simple:

```
def add_time(t1, t2):  
    total = t1.to_seconds() + t2.to_seconds()  
    return MyTime(0, 0, total)
```

When converting back to hours, minutes, and seconds, we use division and modulus operations:

```
hrs = total // 3600  
leftover = total % 3600  
mins = leftover // 60  
secs = leftover % 60
```

This approach simplifies the code and avoids complex logic for handling overflows manually. It also shows how abstraction helps reduce complexity by representing the problem in a simpler numerical form.

GENERALIZATION AND SPECIALIZATION

Mathematicians and computer scientists approach problems differently. Mathematicians often generalize, reducing specific problems into abstract numerical forms. They try to represent many different kinds of problems using general mathematical principles.

Computer scientists, on the other hand, often specialize. They design distinct data types or classes for each kind of problem. For example, instead of representing time as just an integer number of seconds, we define a `MyTime` class that holds hours, minutes, and seconds. This makes the representation more meaningful and reduces the chance of errors.

By specializing, we can prevent invalid operations. For instance, it does not make sense to multiply one time by another, but it is valid to add seconds to a time. Object-oriented programming provides a framework for such specialization by allowing us to define our own data types and restrict operations appropriately.

Comparison Methods

In many cases, we need to compare two objects. For instance, to check if one time occurs after another, we can define a method that performs the comparison.

The basic approach is to compare hours, minutes, and seconds step by step:

```
def after(self, other):
```

```
    if self.hours > other.hours:
```

```
        return True
```

```
    if self.hours < other.hours:
```

```
        return False
```

```
    if self.minutes > other.minutes:
```

```
        return True
```

```
if self.minutes < other.minutes:

    return False

return self.seconds > other.seconds
```

However, using the earlier insight about converting time into total seconds, this can be simplified:

```
def after(self, other):

    return self.to_seconds() > other.to_seconds()
```

This is much cleaner and efficient. The idea is that by representing complex data in a simpler numeric form, comparison and arithmetic operations become straightforward.

OPERATOR OVERLOADING

Operator overloading allows programmers to redefine how standard operators such as `+`, `-`, and `>` work for user-defined types. Python allows this by defining special methods within a class.

For example, to use the `+` operator for time objects, we can define:

```
def __add__(self, other):

    return MyTime(0, 0, self.to_seconds() + other.to_seconds())
```

This lets us write:

```
t3 = t1 + t2
```

Instead of explicitly calling a function. Similarly, we can overload operators for a `Point` class:

```
def __add__(self, other):

    return Point(self.x + other.x, self.y + other.y)
```

For scalar multiplication:

```
def __rmul__(self, n):  
  
    return Point(n * self.x, n * self.y)
```

Operator overloading makes classes easier and more natural to use because their instances behave like built-in types.

POLYMORPHISM

Polymorphism means that a single function or operator can work with multiple types of data as long as those types support the required operations. In Python, this is achieved through dynamic typing, often referred to as “duck typing.” This phrase means that if an object behaves like a duck (it has the necessary methods), Python treats it as one, regardless of its class.

For example, consider the function:

```
def multadd(x, y, z):  
  
    return x * y + z
```

This function can work with numbers or objects like points, as long as the objects define the `__mul__` and `__add__` methods.

We can also define a function like this:

```
def front_and_back(front):  
  
    import copy  
  
    back = copy.copy(front)  
  
    back.reverse()  
  
    print(str(front) + str(back))
```

If we define a reverse method in the Point class, even a point can be used as input to this function:

```
def reverse(self):
```

```
    self.x, self.y = self.y, self.x
```

Then:

```
p = Point(3, 4)
```

```
front_and_back(p)
```

This prints the point and its reversed version. The function works because the object provides the expected methods, even though it is not of a specific type.

Polymorphism gives programs flexibility and reusability. The same code can operate on many different types of objects as long as they implement the required behavior.

EXCEPTIONS

In Python, an **exception** is a runtime error that interrupts the normal flow of a program. Whenever an error occurs, Python creates an **exception object** that contains information about the type of error and the context in which it occurred. If the exception is not handled, the program stops executing and Python prints a **traceback**, showing where the error occurred, ending with a message that describes the exception.

Common examples include:

- **Division by zero:** If a program attempts to divide by zero, a `ZeroDivisionError` is raised. Python stops execution and shows the error type along with a message.
- **Accessing non-existent list items:** Attempting to access an index that does not exist in a list raises an `IndexError`.
- **Modifying immutable objects:** Trying to change an element of a tuple, which is immutable, results in a `TypeError`.

The error message typically contains two parts: the type of error and a description explaining what went wrong.

CATCHING EXCEPTIONS

Sometimes, we want to attempt an operation that might fail but do not want the program to crash. Python provides the try statement to **catch exceptions**. Code that might raise an exception is placed inside a try block, and the program can respond to errors in an except block.

Example:

```
filename = input("Enter a file name: ")

try:

    f = open(filename, "r")

except FileNotFoundError:

    print("There is no file named", filename)
```

Here, if the file does not exist, the program handles the error gracefully instead of crashing.

The try statement can have up to four clauses:

1. **try**: The block of code that may raise an exception.
2. **except**: Code that runs if a specific exception occurs.
3. **else**: Code that runs if no exception occurs in the try block.
4. **finally**: Code that runs no matter what, whether an exception occurs or not.

Example with all clauses:

```
user_input = input("Type a number: ")
```

```
try:
```

```
    number = float(user_input)
```

```
except ValueError:
```

```
    print("You did not enter a number.")
```

```
else:
```

```
    print("The square of your number is", number**2)
```

```
finally:
```

```
    print("Thank you")
```

Best practices: Keep the try block minimal to avoid accidentally catching unintended exceptions. If multiple errors are possible, you can have multiple except clauses for different exception types.

RAISING EXCEPTIONS

Python also allows you to **raise your own exceptions** when your program detects an invalid condition. Raising an exception interrupts the normal flow and passes control to the nearest try block that handles it.

Example:

```
def get_age():
```

```
    age = int(input("Please enter your age: "))
```

```
    if age < 0:
```

```
        raise ValueError(f'{age} is not a valid age')
```

```
    return age
```

Here, if the user enters a negative age, a `ValueError` is raised. The exception propagates through the **call stack** until it reaches a handler. If no handler exists, Python prints a traceback and exits.

Raising exceptions deliberately is useful for **enforcing program constraints** and signaling problems to the caller.

Revisiting Recursion Example with Exceptions

Exceptions can also control complex behaviors like stopping recursion at a safe limit. For example:

```
def recursion_depth(number):  
    print("Recursion depth number", number)  
  
    try:  
        recursion_depth(number + 1)  
    except:  
        print("I cannot go any deeper into this wormhole.")  
  
recursion_depth(0)
```

Here, the exception acts as a stopping condition for recursion without crashing the program.

The finally Clause

In many situations, a program acquires **resources** such as files, network connections, or GUI windows. Even if an error occurs, these resources must be released to avoid resource leaks.

The finally clause ensures that cleanup code is executed **regardless of whether an exception occurs**.

Example with a turtle graphics window:

```
import turtle  
  
import time
```

```
def show_poly():  
  
    try:  
  
        win = turtle.Screen() # Create a resource  
  
        tess = turtle.Turtle()  
  
        n = int(input("How many sides do you want in your polygon? "))  
  
        angle = 360 / n  
  
        for i in range(n):  
  
            tess.forward(10)  
  
            tess.left(angle)  
  
            time.sleep(3)  
  
        finally:  
  
            win.bye() # Close the window no matter what  
  
    show_poly()
```

Even if the user enters invalid input, closes the dialog, or an exception occurs, the finally block ensures that the turtle window is closed properly.

The finally clause is essential for **resource management**, guaranteeing cleanup even in the presence of errors.