Generic Programming in Python

Generic programming in Python refers to writing code that is **not tightly coupled to a specific data type**. Instead, the focus is on building **flexible and reusable code** that can operate on various types without requiring modifications.

Let's see some techniques for generic programming in python

---

1. Generic Types

- Python supports **generic types** using the `typing` module.
- You can define **type-safe functions and classes** that work with a variety of data types.

In [7]:
```python
#### ✅ Example: Generic Function Using `TypeVar`

from typing import TypeVar, List

T = TypeVar('T')  # Define a generic type

def first_element(elements: List[T]) -> T:
    return elements[0]

# Works with int
print(first_element([1, 2, 3]))  # Output: 1

# Works with str
print(first_element(["apple", "banana"]))  # Output: "apple"
```

```
1
apple
```

In [ ]:
```python
# Problems for Practice: Generic Maximum Finder

from typing import TypeVar

T = TypeVar('T')

# A generic method to find the maximum of two values of any type.
def find_max(a: T, b: T) -> T:
    return a if a > b else b

# Example usage
max_int = find_max(10, 5)
print("Max Integer:", max_int)

max_string = find_max("apple", "banana")
print("Max String:", max_string)
```

```
max_double = find_max(3.14, 2.71)
print("Max Double:", max_double)
```

NOTE

Python is dynamically typed, which means it doesn't check types during runtime unless you explicitly do so.

But tools like:

1. mypy
2. pyright (used by VS Code)
3. Linters in IDEs like VS Code, PyCharm

...analyze your code before running it to ensure that you're using types correctly based on your annotations.

Key Difference TypeVar vs Any

Feature	`TypeVar`	`Any`
Defined in	`typing` module	`typing` module
Purpose	Generic type placeholder	Accept any type without checking
Type safety	✅ Yes	❌ No
Use case	Reusable, type-safe code	Dynamic, flexible code
Type consistency	✅ Enforced	❌ Not enforced
Runtime effect	No effect (only for checkers)	No effect (only for checkers)

2. Duck Typing

Python uses duck typing, which means:

- "If it walks like a duck and quacks like a duck, it's a duck."
- An object's behavior (methods/properties) is more important than its type.

In [9]:
```
class PayPal:
    def pay(self, amount):
        print(f"Paying ₹{amount} using PayPal")

class CreditCard:
    def pay(self, amount):
        print(f"Paying ₹{amount} using Credit Card")
```

In [11]:
```
# client code
def process_payment(payment_method, amount):
```

```
    payment_method.pay(amount)
```

In [12]:
```
paypal = PayPal()
credit_card = CreditCard()

process_payment(paypal, 500)
process_payment(credit_card, 1000)
```

```
Paying ₹500 using PayPal
Paying ₹1000 using Credit Card
```

3.Function Overloading:

While Python does not support traditional function overloading (defining multiple functions with the same name but different parameter types or counts), you can achieve similar behavior using default parameter values or variable arguments.

---

4. Polymorphism :

Python supports polymorphism, allowing different objects to be treated as instances of the same class through inheritance and method overriding.

This enables writing generic code that operates on objects of different types in a uniform way.

In [8]:
```
# Base interface
class PaymentMethod:
    def pay(self, amount):
        raise NotImplementedError

# Subclasses
class CreditCard(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} using Credit Card")

class PayPal(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} using PayPal")

class UPI(PaymentMethod):
    def pay(self, amount):
        print(f"Paid ₹{amount} using UPI")

# Client code using polymorphism
def process_payment(payment: PaymentMethod, amount):
    payment.pay(amount)

# Usage
process_payment(CreditCard(), 1000)
```

```
process_payment(PayPal(), 500)
process_payment(UPI(), 200)
```

```
Paid ₹1000 using Credit Card
Paid ₹500 using PayPal
Paid ₹200 using UPI
```

Benefits of Generic Programming

- Code reusability across different data types.
- Cleaner and maintainable code.
- Python makes it easy to write generic and type-safe code using features like:

1. TypeVar, Generic, and typing module
2. Duck typing for flexibility and runtime polymorphism

In [ ]: