# Python and OOPS

S1 MCA

# Advantages of OOP

Provides a clear structure to programs

Makes code easier to maintain, reuse, and debug

Helps keep your code DRY (**Don't Repeat Yourself**)

Allows you to build reusable applications with less code

# What are Classes and Objects?

- **What is a Class?**
- A **class** is a blueprint or template for creating objects.
  It defines the **attributes (data)** and **methods (functions)** that describe the behavior of an object.
- Think of a class as a *blueprint of a house* and the object as an *actual house built from that blueprint*.

```
>>>class Student:
        name = "John"
        age = 20
```

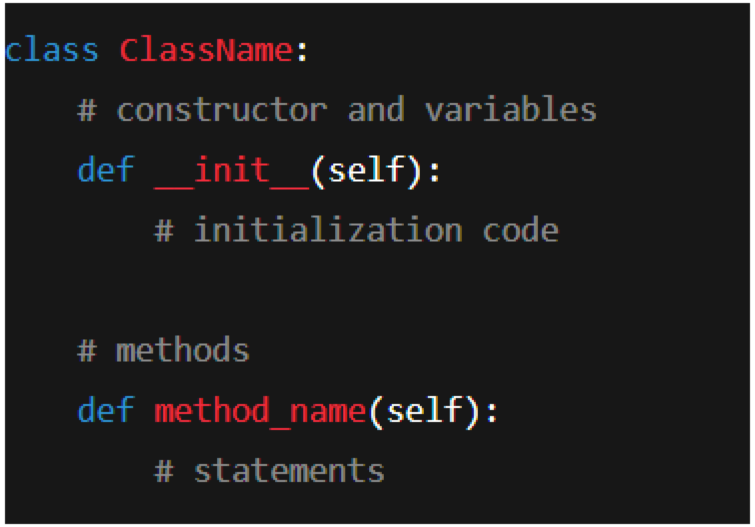
- Student → class name
- name and age → class attributes

# What is an Object?

- An **object** is an instance of a class.
- It represents one specific example of the class.

```
>>>
class Student:
    name = “Devin"
    age = 20
# Create object
s1 = Student()
# Access attributes
print(s1.name)
print(s1.age)
```

Syntax



```
class ClassName:
    # constructor and variables
    def __init__(self):
        # initialization code

    # methods
    def method_name(self):
        # statements
```

Here s1 is an **object** of the class Student.

## Delete Objects

- You can delete objects by using the del keyword:

```
>>>
 del S1
```

## Multiple Objects

- You can create multiple objects from the same class:

```
S1 = Student()
S2 = Student()


Print(S1.name)
Print(S2.name)
```

**Each object is independent and has its own copy of the class properties.**

## The pass Statement

- class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

```
>>>

class Person:
  pass
```

## The __init__() Method

- All classes have a built-in method called __init__(), which is always executed when the class is being initiated.
- The __init__() method is used to assign values to object properties, or to perform operations that are necessary when the object is being created.

```
>>>

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

p1 = Person("Emil", 36)

print(p1.name)
print(p1.age)
```

**The __init__() method is called automatically every time the class is being used to create a new object.**

## Why Use __init__()?

- Without the __init__() method, you would need to set properties manually for each object:

**>>>**

```
class Person:
  pass

p1 = Person()
p1.name = "Tobias"
p1.age = 25

print(p1.name)
print(p1.age)
```

- Using __init__() makes it easier to create objects with initial values:

```
>>>

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

p1 = Person("Linus", 28)

print(p1.name)
print(p1.age)
```

## Default Values in __init__()

- You can also set default values for parameters in the __init__() method:

```
>>>

class Person:
  def __init__(self, name, age=18):
    self.name = name
    self.age = age

p1 = Person("Emil")
p2 = Person("Tobias", 25)

print(p1.name, p1.age)
print(p2.name, p2.age)
```

## Multiple Parameters

- The __init__() method can have as many parameters as you need:

**>>>**

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

p1 = Person("Linus", 30, "Oslo", "Norway")

print(p1.name)
print(p1.age)
print(p1.city)
print(p1.country)
```

## The self Parameter

- The self parameter is a reference to the current instance of the class.
- It is used to access properties and methods that belong to the class.

```
>>>

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

  def greet(self):
    print("Hello, my name is " + self.name)

p1 = Person("Emil", 25)
p1.greet()
```

**The self parameter must be the first parameter of any method in the class.**

- Without self, Python would not know which object's properties you want to access:
- The self parameter links the method to the specific object:

```
>>>

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

  def printname(self):
    print(self.name)

p1 = Person("Tobias")
p2 = Person("Linus")

p1.printname()
p2.printname()
```

- It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any method in the class:

```
>>>

class Person:
  def __init__(myobject, name, age):
    myobject.name = name
    myobject.age = age

  def greet(abc):
    print("Hello, my name is " + abc.name)

p1 = Person("Emil", 36)
p1.greet()
```

**Here Used the words *myobject* and *abc* instead of *self*:**

## Accessing Properties with self

- You can access any property of the class using self:

```
>>>

class Car:
  def __init__(self, brand, model, year):
    self.brand = brand
    self.model = model
    self.year = year

  def display_info(self):
    print(f"{self.year} {self.brand} {self.model}")

car1 = Car("Toyota", "Corolla", 2020)
car1.display_info()
```

## Calling Methods with self

- Call one method from another method using self:

```
>>>

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

  def greet(self):
    return "Hello, " + self.name

  def welcome(self):
    message = self.greet()
    print(message + "! Welcome to our website.")

p1 = Person("Tobias")
p1.welcome()
```

## Class Properties

- Properties are variables that belong to a class. They store data for each object created from the class.

**>>>**

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

p1 = Person("Emil", 36)

print(p1.name)
print(p1.age)
```

# Access Properties

- You can access object properties using dot notation:

```
>>>

class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

car1 = Car("Toyota", "Corolla")

print(car1.brand)
print(car1.model)
```

# Modify Properties

- You can modify the value of properties on objects:

```
>>>

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

p1 = Person("Tobias", 25)
print(p1.age)

p1.age = 26
print(p1.age)
```

# Delete Properties

- You can delete properties from objects using the del keyword:

```
>>>

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

p1 = Person("Linus", 30)

del p1.age

print(p1.name) # This works
# print(p1.age) # This would cause an error
```

# Class Properties vs Object Properties

- Properties defined inside __init__() belong to each object (instance properties).
- Properties defined outside methods belong to the class itself (class properties) and are shared by all objects

```
>>>class Person:
  species = "Human" # Class property

  def __init__(self, name):
    self.name = name # Instance property

p1 = Person("Emil")
p2 = Person("Tobias")

print(p1.name)
print(p2.name)
print(p1.species)
print(p2.species)
```

# Modifying Class Properties

- When you modify a class property, it affects all objects:

```
>>>
class Person:
  lastname = ""

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

p1 = Person("Linus")
p2 = Person("Emil")

Person.lastname = "Refsnes"

print(p1.lastname)
print(p2.lastname)
```

# Add New Properties

- You can add new properties to existing objects:

```
>>>

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

p1 = Person("Tobias")

p1.age = 25
p1.city = "Oslo"

print(p1.name)
print(p1.age)
print(p1.city)
```

# Python Class Methods

- Methods are functions that belong to a class. They define the behavior of objects created from the class.

```
>>>

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

  def greet(self):
    print("Hello, my name is " + self.name)

p1 = Person("Emil")
p1.greet()
```

- **Note:** All methods must have self as the first parameter.

# Methods with Parameters

- Methods can accept parameters just like regular functions:

```
>>>

class Calculator:
  def add(self, a, b):
    return a + b

  def multiply(self, a, b):
    return a * b

calc = Calculator()
print(calc.add(5, 3))
print(calc.multiply(4, 7))
```

# Methods Accessing Properties

- Methods can access and modify object properties using self:

```
>>>

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

  def get_info(self):
    return f"{self.name} is {self.age} years old"

p1 = Person("Tobias", 28)
print(p1.get_info())
```

# Methods Modifying Properties

- Methods can modify the properties of an object:

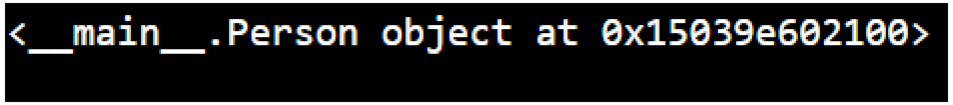
```
>>>

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

  def celebrate_birthday(self):
    self.age += 1
    print(f"Happy birthday! You are now {self.age}")

p1 = Person("Linus", 25)
p1.celebrate_birthday()
p1.celebrate_birthday()
```

# The __str__() Method

- The __str__() method is a special method that controls what is returned when the object is printed:

**>>>**Without the __str__() method:

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

p1 = Person("Emil", 36)
print(p1)
```

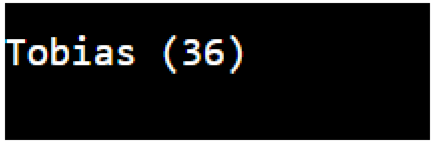


```
<__main__.Person object at 0x15039e602100>
```

# The __str__() Method

**>>>**With the __str__() method:

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

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

p1 = Person("Tobias", 36)
print(p1)
```

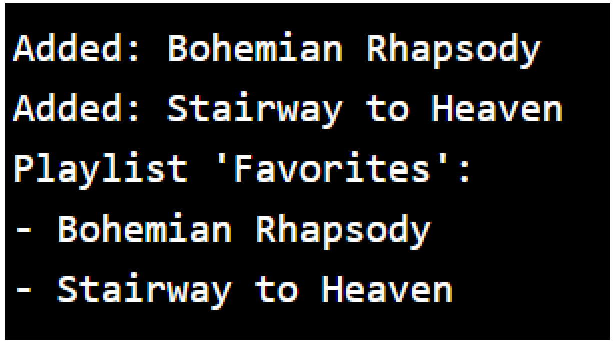


```
Tobias (36)
```

# Multiple Methods

- A class can have multiple methods that work together:

```
class Playlist:
  def __init__(self, name):
    self.name = name
    self.songs = []

  def add_song(self, song):
    self.songs.append(song)
    print(f"Added: {song}")

  def remove_song(self, song):
    if song in self.songs:
      self.songs.remove(song)
      print(f"Removed: {song}")

  def show_songs(self):
    print(f"Playlist '{self.name}':")
    for song in self.songs:
      print(f"- {song}")

my_playlist = Playlist("Favorites")
my_playlist.add_song("Bohemian Rhapsody")
my_playlist.add_song("Stairway to Heaven")
my_playlist.show_songs()
```



```
Added: Bohemian Rhapsody
Added: Stairway to Heaven
Playlist 'Favorites':
- Bohemian Rhapsody
- Stairway to Heaven
```

## Delete Methods

- You can delete methods from a class using the del keyword:

```
>>>

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

  def greet(self):
    print("Hello!")

p1 = Person("Emil")

del Person.greet

p1.greet() # This will cause an error
```

## Python Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

## Create a Parent Class

- Any class can be a parent class, so the syntax is the same as creating any other class:

## Create a Child Class

- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:
- Create a class named Student, which will inherit the properties and methods from the Person class:

```
>>>

class Student(Person):
  pass
```

- Now the Student class has the same properties and methods as the Person class.
- Use the Student class to create an object, and then execute the printname method:

**>>>**

```
x = Student("Mike", "Olsen")
x.printname()
```

## Add the __init__() Function

- So far we have created a child class that inherits the properties and methods from its parent.
- We want to add the __init__() function to the child class (instead of the pass keyword).
- **Note:** The __init__() function is called automatically every time the class is being used to create a new object.
- Add the __init__() function to the Student class:

```
>>>

class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

- When you add the __init__() function, the child class will no longer inherit the parent's __init__() function.
- **Note:** The child's __init__() function **overrides** the inheritance of the parent's __init__() function.

- To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:

**>>>**

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the __init__() function.

**Use the super() Function**

- Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

- By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

## Add Properties

- Add a property called graduationyear to the Student class:

```
>>>

class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

- In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function:

```
>>>

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

## Add Methods

- Add a method called welcome to the Student class:

**>>>**

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)
```

## Python Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

## Function Polymorphism

- An example of a Python function that can be used on different objects is the len() function.
- For strings len() returns the number of characters:
- For tuples len() returns the number of items in the tuple:
- For dictionaries len() returns the number of key/value pairs in the dictionary:
-

**Class Polymorphism**

- Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.
- For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move():

- Different classes with the same method:
- Look at the for loop at the end. Because of polymorphism we can execute the same method for all three classes.

**>>>**

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Drive!")

class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Sail!")

class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747")     #Create a Plane object

for x in (car1, boat1, plane1):
  x.move()
```

# Inheritance Class Polymorphism

- What about classes with child classes with the same name? Can we use polymorphism there?
- Yes. If we use the example above and make a parent class called Vehicle, and make Car, Boat, Plane child classes of Vehicle, the child classes inherits the Vehicle methods, but can override them:
-

  Create a class called Vehicle and make Car, Boat, Plane child classes of Vehicle:
-

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747")     #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

- Child classes inherits the properties and methods from the parent class.
- In the example above you can see that the Car class is empty, but it inherits brand, model, and move() from Vehicle.
- The Boat and Plane classes also inherit brand, model, and move() from Vehicle, but they both override the move() method.
- Because of polymorphism we can execute the same method for all classes.

## Python Encapsulation

- Encapsulation is about protecting data inside a class.
- It means keeping data (properties) and methods together in a class, while controlling how the data can be accessed from outside the class.
- This prevents accidental changes to your data and hides the internal details of how your class works.

## Private Properties

- In Python, you can make properties private by using a double underscore __ prefix:

- Create a private class property named __age:

**>>>**

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.__age = age # Private property

p1 = Person("Emil", 25)
print(p1.name)
print(p1.__age) # This will cause an error
```

# Get Private Property Value

- To access a private property, you can create a getter method:

**>>>**

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

  def get_age(self):
    return self.__age

p1 = Person("Tobias", 25)
print(p1.get_age())
```

## Set Private Property Value

- To modify a private property, you can create a setter method.
- The setter method can also validate the value before setting it:

**>>>**

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

  def get_age(self):
    return self.__age

  def set_age(self, age):
    if age > 0:
      self.__age = age
    else:
      print("Age must be positive")

p1 = Person("Tobias", 25)
print(p1.get_age())

p1.set_age(26)
print(p1.get_age())
```

# Why Use Encapsulation?

- Encapsulation provides several benefits:
- **Data Protection:** Prevents accidental modification of data
- **Validation:** You can validate data before setting it
- **Flexibility:** Internal implementation can change without affecting external code
- **Control:** You have full control over how data is accessed and modified

- Use encapsulation to protect and validate data:

```
class Student:
  def __init__(self, name):
    self.name = name
    self.__grade = 0

  def set_grade(self, grade):
    if 0 <= grade <= 100:
      self.__grade = grade
    else:
      print("Grade must be between 0 and 100")

  def get_grade(self):
    return self.__grade

  def get_status(self):
    if self.__grade >= 60:
      return "Passed"
    else:
      return "Failed"

student = Student("Emil")
student.set_grade(85)
print(student.get_grade())
print(student.get_status())
```

## Protected Properties

Python also has a convention for protected properties using a single underscore _ prefix:

- Create a protected property:

**>>>**

```
class Person:
  def __init__(self, name, salary):
    self.name = name
    self._salary = salary # Protected property

p1 = Person("Linus", 50000)
print(p1.name)
print(p1._salary) # Can access, but shouldn't
```

A single underscore _ is just a convention. It tells other programmers that the property is intended for internal use, but Python doesn't enforce this restriction.

## Introduction to Operator Overloading

- In Python, operators like +, -, >, and == can behave differently for user-defined objects.
- This is done using Special Methods (also called Magic Methods or Dunder Methods — Double Underscore Methods).

```
>>>

a = 5

b = 10

print(a + b)   # calls a.__add__(b)
```

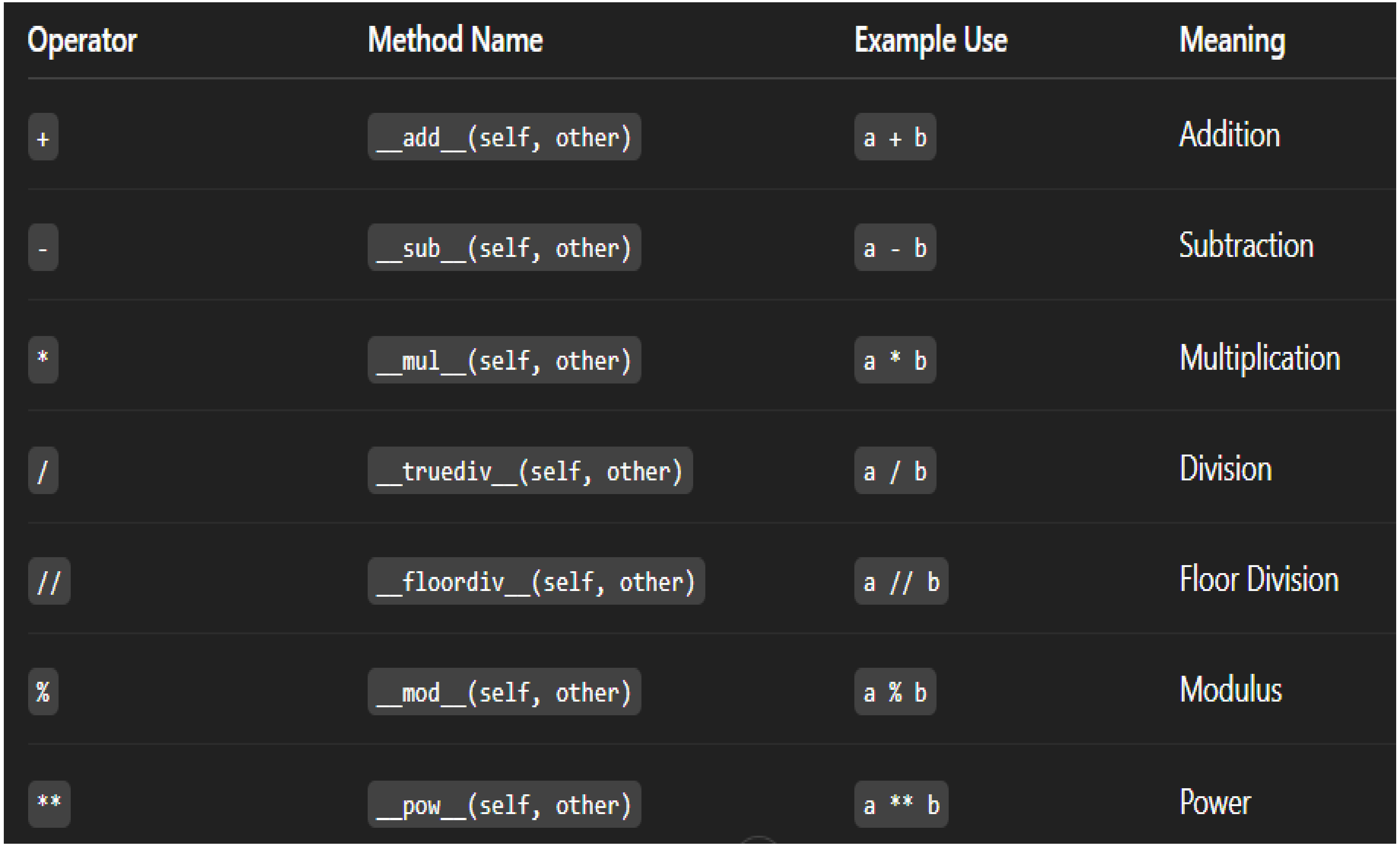
So + is just a shortcut for calling the __add__() method internally.

## What is Operator Overloading?

Operator Overloading means **defining how operators behave** for objects of a user-defined class.

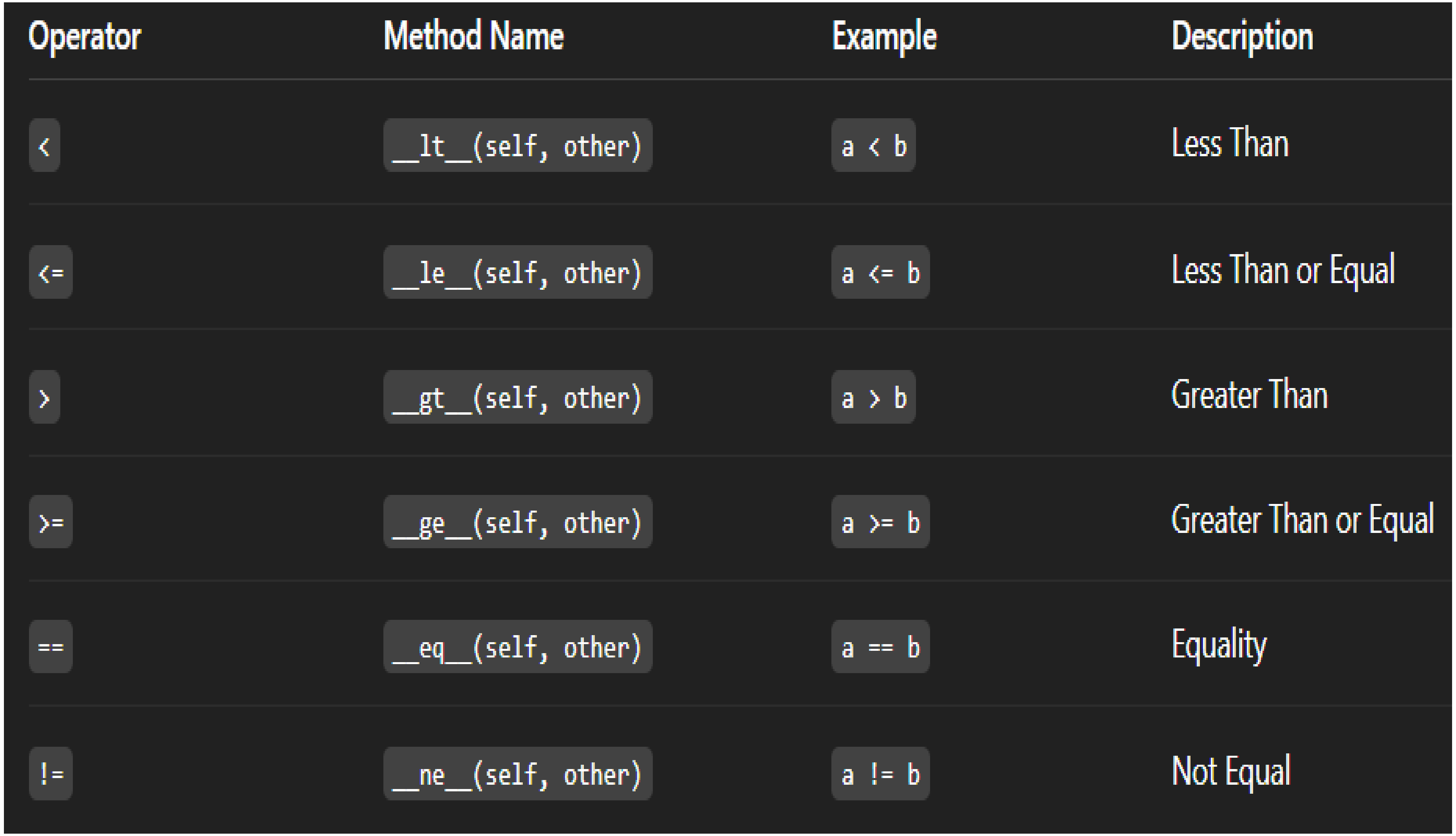
```
>>>
class Example:
    def __init__(self, x):
        self.x = x

    def __add__(self, other):  # Overloading '+'
        return Example(self.x + other.x)

a = Example(10)
b = Example(20)
c = a + b    # internally calls a.__add__(b)
print(c.x)
```

# Common Magic Methods for Operator Overloading



| Operator | Method Name | Example Use | Meaning |
|---|---|---|---|
| `+` | `__add__(self, other)` | `a + b` | Addition |
| `-` | `__sub__(self, other)` | `a - b` | Subtraction |
| `*` | `__mul__(self, other)` | `a * b` | Multiplication |
| `/` | `__truediv__(self, other)` | `a / b` | Division |
| `//` | `__floordiv__(self, other)` | `a // b` | Floor Division |
| `%` | `__mod__(self, other)` | `a % b` | Modulus |
| `**` | `__pow__(self, other)` | `a ** b` | Power |

# Comparison Operators



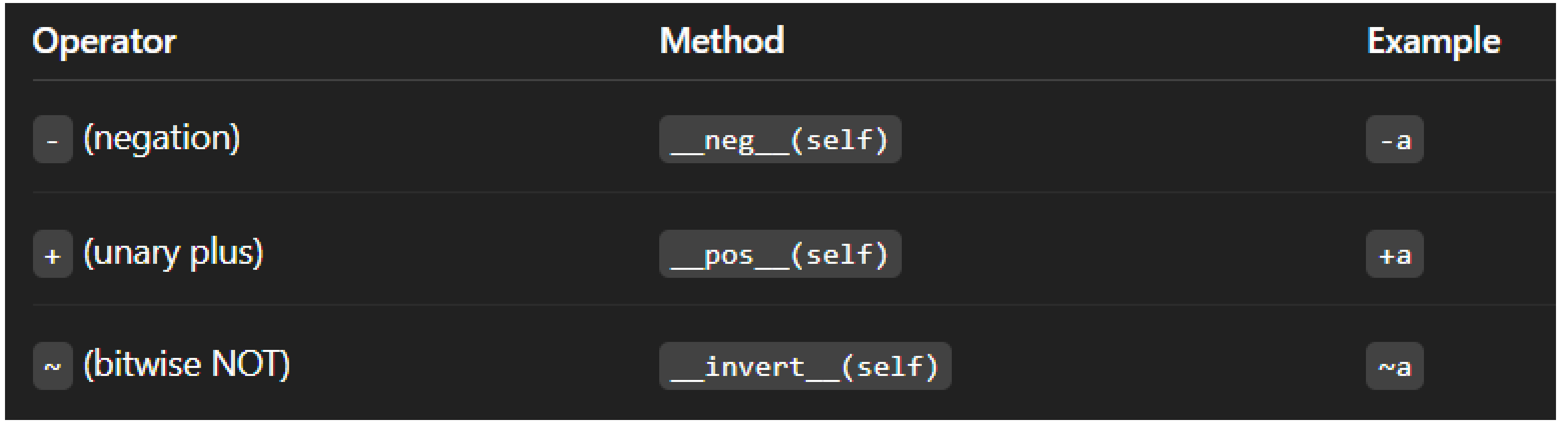
| Operator | Method Name | Example | Description |
|---|---|---|---|
| `<` | `__lt__(self, other)` | `a < b` | Less Than |
| `<=` | `__le__(self, other)` | `a <= b` | Less Than or Equal |
| `>` | `__gt__(self, other)` | `a > b` | Greater Than |
| `>=` | `__ge__(self, other)` | `a >= b` | Greater Than or Equal |
| `==` | `__eq__(self, other)` | `a == b` | Equality |
| `!=` | `__ne__(self, other)` | `a != b` | Not Equal |

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def __lt__(self, other):  # Overload '<'
        return self.area() < other.area()

r1 = Rectangle(5, 10)
r2 = Rectangle(6, 8)

if r1 < r2:
    print("Rectangle 1 is smaller.")
else:
    print("Rectangle 2 is smaller.")
```

```python
class Time:
    def __init__(self, h, m, s):
        self.h = h
        self.m = m
        self.s = s

    def __add__(self, other):
        total_s = self.s + other.s
        total_m = self.m + other.m + total_s // 60
        total_h = self.h + other.h + total_m // 60

        total_s %= 60
        total_m %= 60
        return Time(total_h, total_m, total_s)

    def display(self):
        print(f"{self.h:02d}:{self.m:02d}:{self.s:02d}")

t1 = Time(2, 45, 50)
t2 = Time(1, 30, 20)
t3 = t1 + t2
t3.display()
```

# Unary Operator Overloading

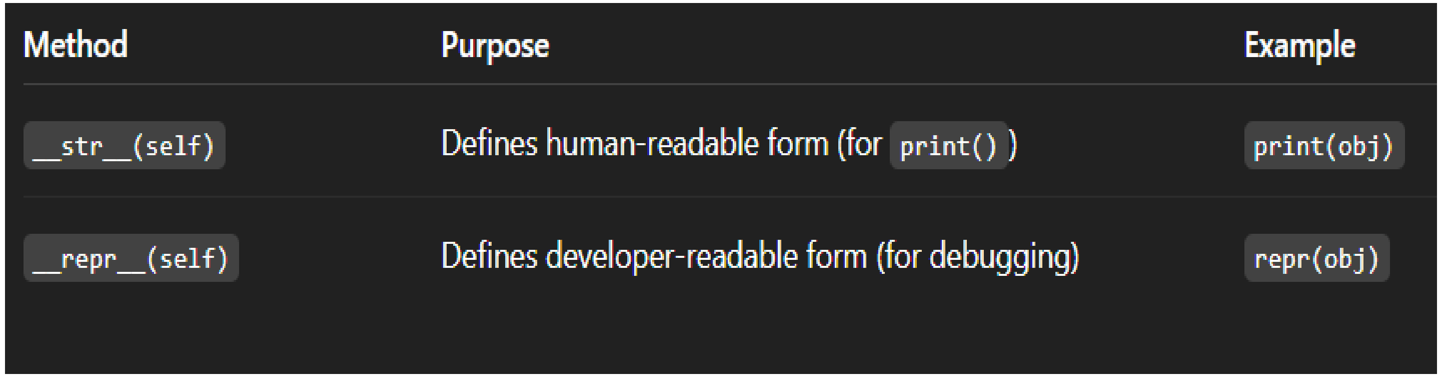
```
>>>
class Number:
    def __init__(self, value):
        self.value = value

    def __neg__(self):
        return Number(-self.value)


n = Number(5)
m = -n
print(m.value)
```



| Operator | Method | Example |
|---|---|---|
| `-` (negation) | `__neg__(self)` | `-a` |
| `+` (unary plus) | `__pos__(self)` | `+a` |
| `~` (bitwise NOT) | `__invert__(self)` | `~a` |

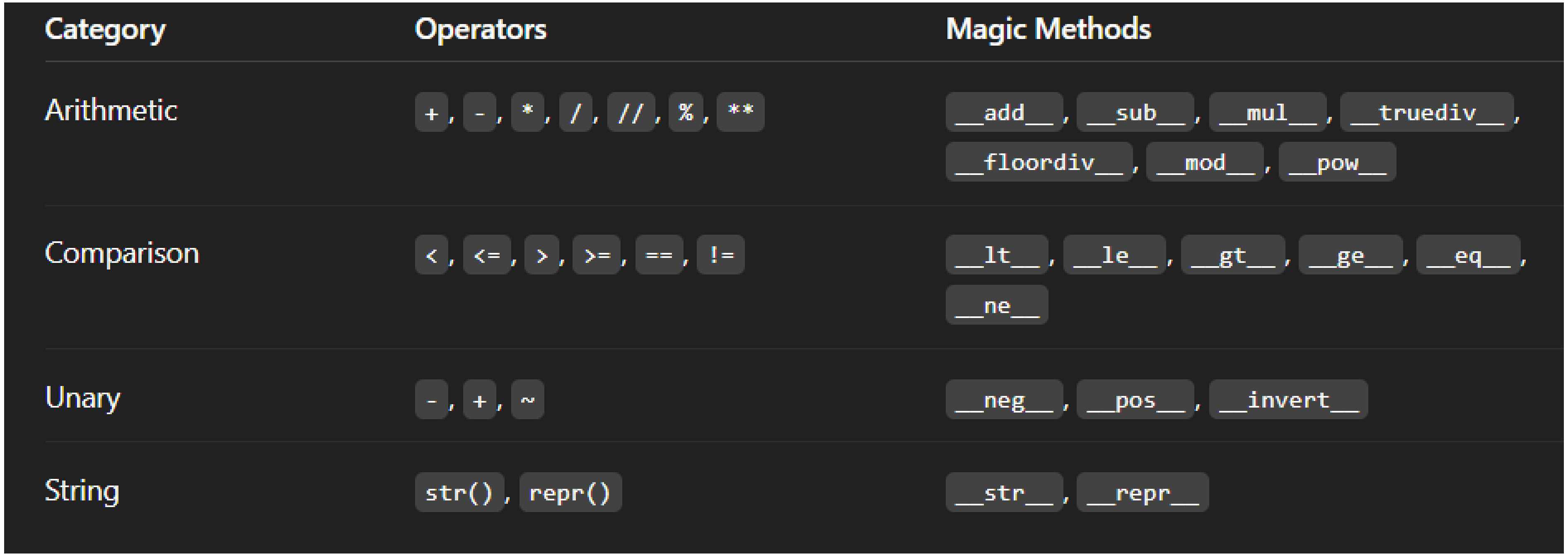
# String Representation Operators

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

    def __str__(self):
        return f"Name: {self.name}, Age: {self.age}"

s = Student("John", 21)
print(s)
```



| Method | Purpose | Example |
|---|---|---|
| `__str__(self)` | Defines human-readable form (for `print()`) | `print(obj)` |
| `__repr__(self)` | Defines developer-readable form (for debugging) | `repr(obj)` |

- ✅ Operator overloading lets your class **behave like built-in types**
  ✅ You implement it using **special (dunder) methods**
  ✅ Each operator has a corresponding **method name**
  ✅ Makes code more readable and object-oriented



| Category | Operators | Magic Methods |
|---|---|---|
| Arithmetic | `+`, `-`, `*`, `/`, `//`, `%`, `**` | `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__pow__` |
| Comparison | `<`, `<=`, `>`, `>=`, `==`, `!=` | `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__` |
| Unary | `-`, `+`, `~` | `__neg__`, `__pos__`, `__invert__` |
| String | `str()`, `repr()` | `__str__`, `__repr__` |

## Python Try Except

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The else block lets you execute code when there is no error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

## Exception Handling

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
- These exceptions can be handled using the try statement:

- The try block will generate an exception, because x is not defined:

**>>>**

```
try:
  print(x)
except:
  print("An exception occurred")
```

- Since the try block raises an error, the except block will be executed.
- Without the try block, the program will crash and raise an error:

## Many Exceptions

- You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:
- Print one message if the try block raises a NameError and another for other errors:

**>>>**

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

## Else

- You can use the else keyword to define a block of code to be executed if no errors were raised:
- In this example, the try block does not generate any error:

**>>>**

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

## Finally

- The finally block, if specified, will be executed regardless if the try block raises an error or not.

**>>>**

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

## Raise an exception

- As a Python developer you can choose to throw an exception if a condition occurs.
- To throw (or raise) an exception, use the raise keyword.

**>>>**

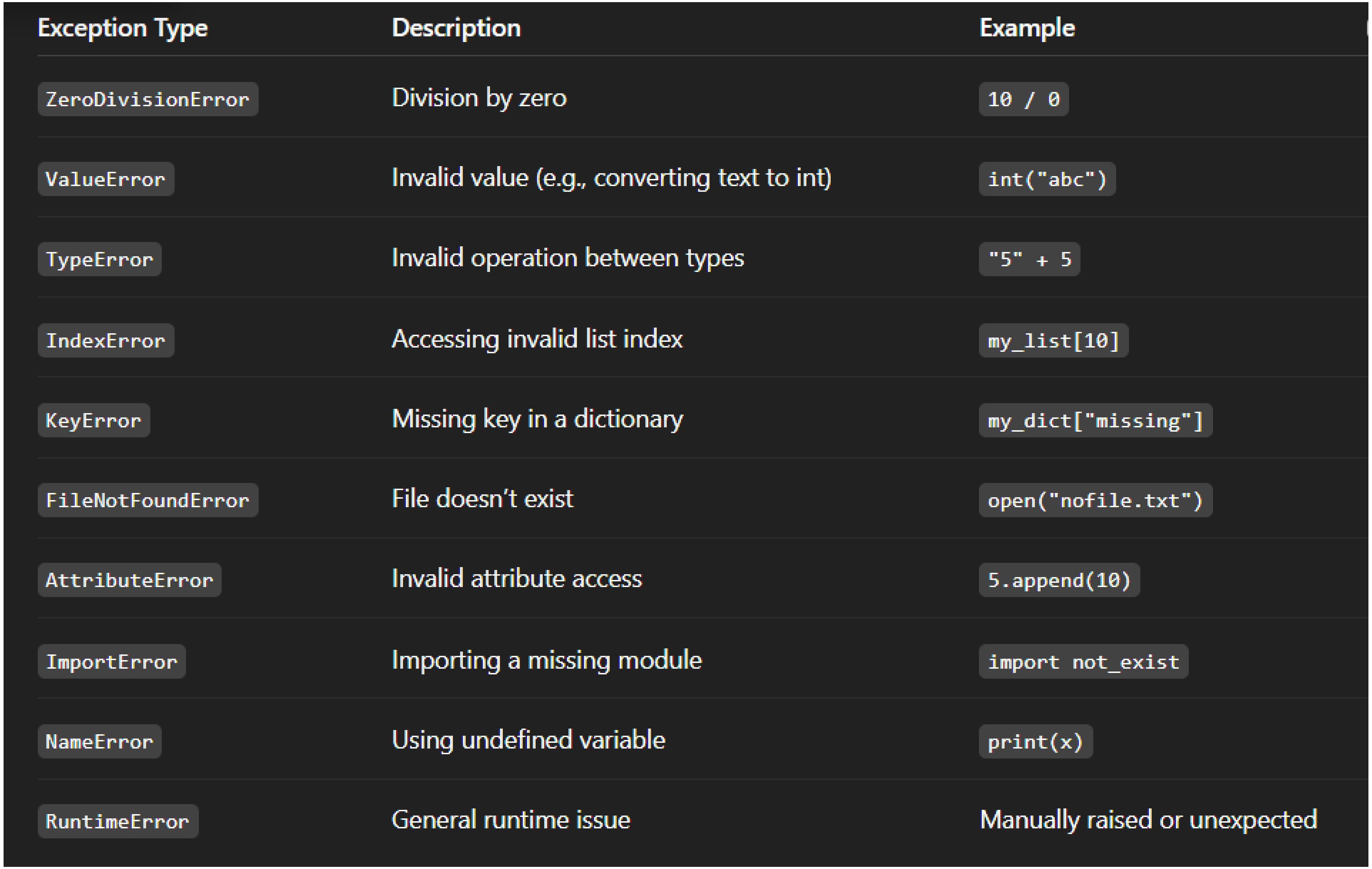
```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

- The raise keyword is used to raise an exception.
- You can define what kind of error to raise, and the text to print to the user.

- Raise a TypeError if x is not an integer:

```
>>>
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

# 🚫 Common Python Exceptions (Errors)



| Exception Type | Description | Example |
|---|---|---|
| `ZeroDivisionError` | Division by zero | `10 / 0` |
| `ValueError` | Invalid value (e.g., converting text to int) | `int("abc")` |
| `TypeError` | Invalid operation between types | `"5" + 5` |
| `IndexError` | Accessing invalid list index | `my_list[10]` |
| `KeyError` | Missing key in a dictionary | `my_dict["missing"]` |
| `FileNotFoundError` | File doesn't exist | `open("nofile.txt")` |
| `AttributeError` | Invalid attribute access | `5.append(10)` |
| `ImportError` | Importing a missing module | `import not_exist` |
| `NameError` | Using undefined variable | `print(x)` |
| `RuntimeError` | General runtime issue | Manually raised or unexpected |

## Catching Multiple Exceptions Together

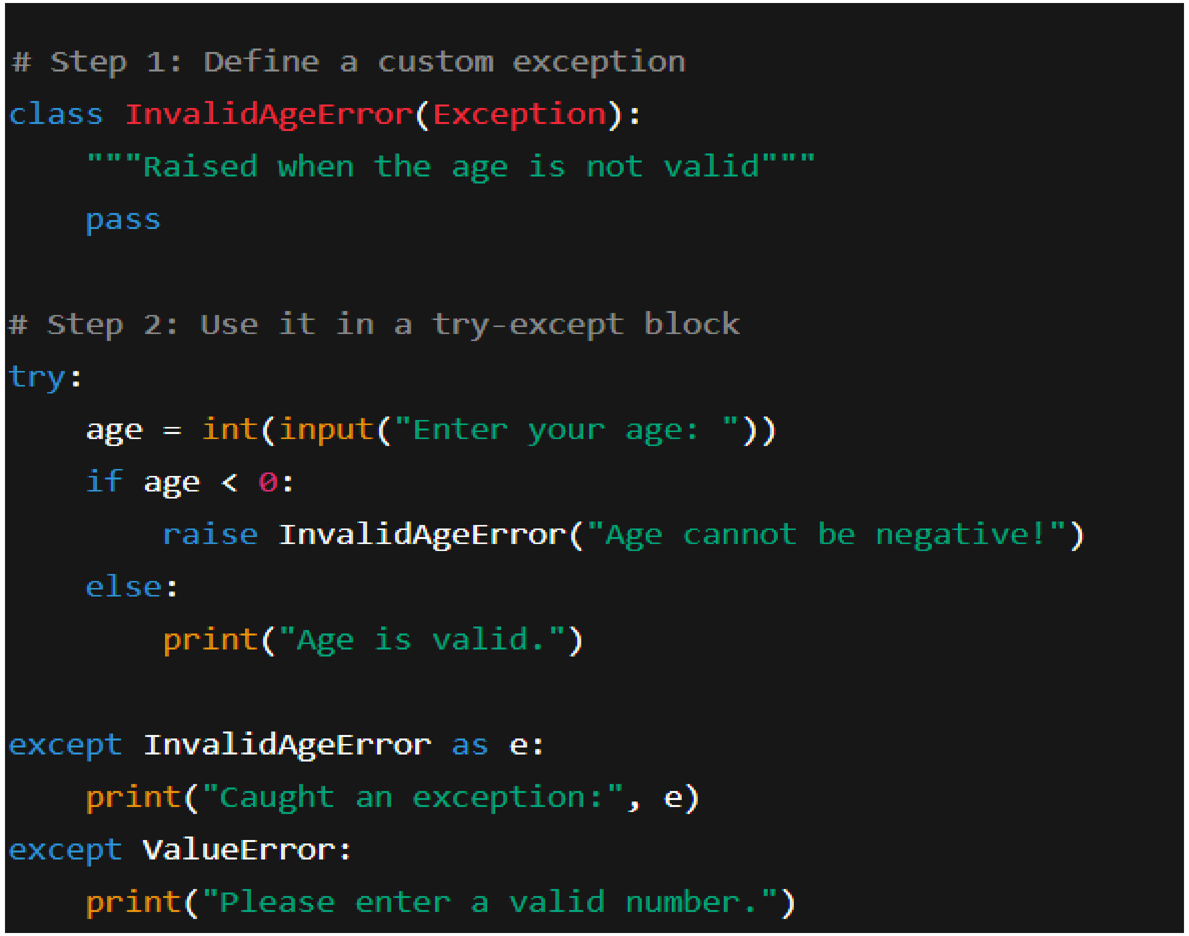
**>>>**

```
try:
    x = int("abc")
    y = 10 / 0

except (ValueError, ZeroDivisionError) as e:
    print(f"An error occurred: {e}")
```

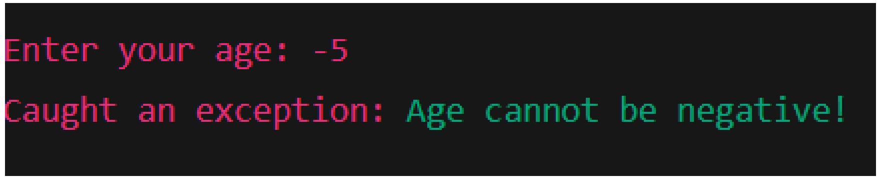
# User-Defined Exception

A **user-defined exception** is a **custom error type** that you create yourself when you want to handle a **specific kind of error** that isn’t covered by Python’s built-in exceptions.

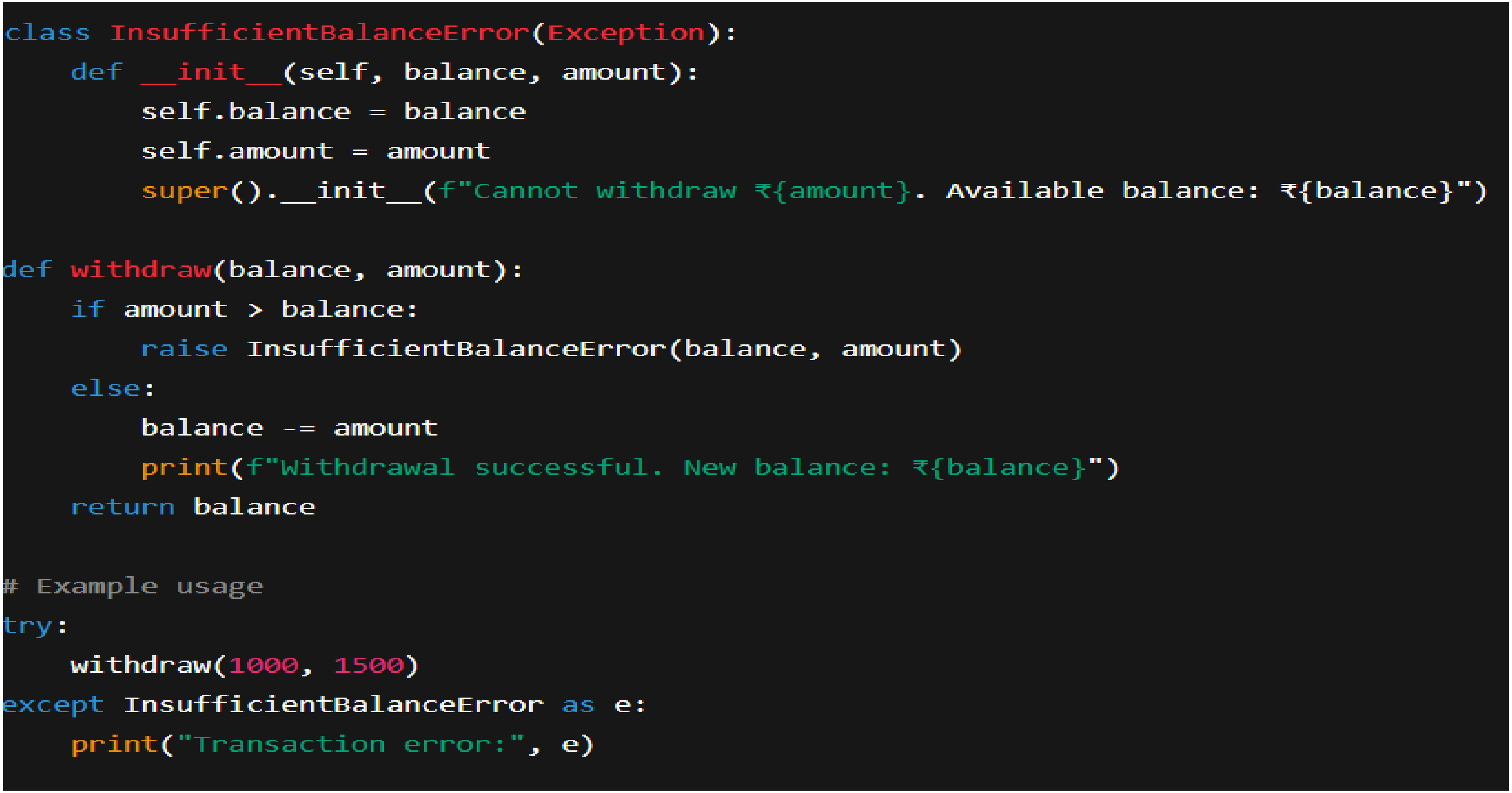
You define it by creating a **class that inherits from Exception** (or a subclass of it).



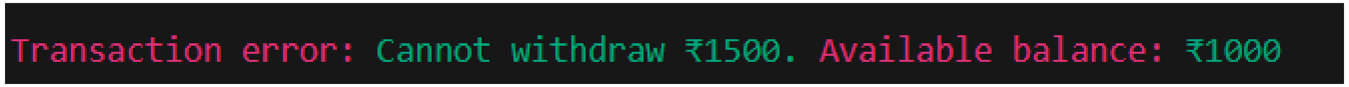
```
# Step 1: Define a custom exception
class InvalidAgeError(Exception):
    """Raised when the age is not valid"""
    pass

# Step 2: Use it in a try-except block
try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise InvalidAgeError("Age cannot be negative!")
    else:
        print("Age is valid.")

except InvalidAgeError as e:
    print("Caught an exception:", e)
except ValueError:
    print("Please enter a valid number.")
```



```
Enter your age: -5
Caught an exception: Age cannot be negative!
```



```python
class InsufficientBalanceError(Exception):
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        super().__init__(f"Cannot withdraw ₹{amount}. Available balance: ₹{balance}")

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientBalanceError(balance, amount)
    else:
        balance -= amount
        print(f"Withdrawal successful. New balance: ₹{balance}")
    return balance

# Example usage
try:
    withdraw(1000, 1500)
except InsufficientBalanceError as e:
    print("Transaction error:", e)
```



```
Transaction error: Cannot withdraw ₹1500. Available balance: ₹1000
```