Intermediate object-oriented programming

Joachim Vandekerckhove

Input validation

Input Validation

- Input validation is the process of checking if input data is valid before using it.
- It helps ensure that your program runs correctly and securely.
- Common types of input validation include range checking, type checking, format checking, and existence checking.

Benefits of Input Validation

- Helps prevent unexpected behavior.
- Improves code readability and maintainability.
- Reduces security risks.
- Provides better user experience by giving informative error messages.

Common Types of Input Validation

- Range checking: Ensures input falls within acceptable range of values.
- Type checking: Ensures input is of the expected type.
- Format checking: Ensures input conforms to a specific format (e.g., phone numbers or email addresses).
- Existence checking: Ensures input is not empty or null.

Basic Input Validation Example

```python
class BankAccount:
    def __init__(self, owner, balance):
        if balance < 0:
            raise ValueError("Balance cannot be negative.")
        self.owner = owner
        self.balance = balance
    # ...
```

Handling Invalid Inputs Gracefully

- Provide informative error messages to the user.
- Log error messages for developers to review.
- Raise exceptions with descriptive error messages.
- Ensure that the program does not continue with invalid input.

Input Validation in the Context of OOP

- Input validation can be incorporated into methods and class constructors.
- It's a good practice to perform input validation as close to the source of input as possible.
- Encapsulation allows you to define rules for valid state and prevent invalid input from corrupting the state of the object.

Corruption

Corrupted State

A corrupted state is a situation where an object's data is in an inconsistent or invalid state, often caused by an error in the program or a violation of the object's invariants.

- An object is in a corrupted state when its internal data is inconsistent with the object's intended behavior
- A corrupted state can lead to bugs and errors in the program
- Preventing corrupted states is an important part of writing correct, reliable code

Example: Workflow that leads to a Corrupted State

Consider a bank account object, with a balance attribute and a deposit and withdraw method.

```python
class BankAccount:

    def __init__(self, balance):
        if balance < 0:
            raise ValueError("Balance must be positive.")
        self.balance = balance

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

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Overdrawing not allowed!")
        self.balance -= amount
```

Example: Workflow that leads to a Corrupted State

Suppose we have the following workflow:

1. Create a new bank account with initial balance of 100 dollars
2. Make a deposit of -50 dollars
3. That's arguably not a deposit
4. This workflow violates the invariants of the bank account object and results in a corrupted state, where the balance is negative.

Example: External Function Bypassing Logger Method

Let's add a "history" attribute and a "logger" method to keep track of all the transactions made on the account:

```python
class BankAccount:

    def __init__(self, balance):
        self.balance = balance
        self.history = []

    def deposit(self, amount):
        self.balance += amount
        self.history.append(('deposit', amount))

    def withdraw(self, amount):
        self.balance -= amount
        self.history.append(('withdraw', amount))

    def logger(self):
        for transaction in self.history:
            print(transaction)
```

Example: External Function Bypassing Logger Method

Now suppose an *external* function edits the balance attribute of the BankAccount object directly, bypassing the deposit and withdraw methods and the logger method:

```python
def yoink(account):
    account.balance -= 100
```

This bypasses the logger method and leads to a corrupted state where the transaction history no longer reflects the actual transactions made on the account.

Private vs. Public

Private vs. Public in OOP

Public methods and attributes are accessible from outside the class

Private methods and attributes are only accessible within the class

```python
class MyClass:

    def __init__(self):
        self.public_attr = 0
        self.__private_attr = 1

    def public_method(self):
        print("This is a public method")
        self.__private_method()

    def __private_method(self):
        print("This is a private method")
```

Private vs Public Methods and Attributes

- By convention, two leading underscores indicates a “private” method or attribute
- Private methods and attributes are intended for internal use only, and should not be accessed from outside the class
- Public methods and attributes can be freely accessed from outside the class
- In Python, there are no *truly* private methods or attributes

Private Attributes in the BankAccount Class

```python
class BankAccount:
    def __init__(self, initial_balance):
        if initial_balance < 0:
            raise ValueError("Initial balance cannot be <0")
        self.__balance = initial_balance

    def get_balance(self):
        return self.__balance
```

- The ‘__balance‘ attribute in ‘BankAccount‘ is now private
- It is accessed using the ‘get_balance()‘ method, which is public
- External code shouldn’t modify the ‘__balance‘ attribute directly
- This ensures that the balance is always valid and prevents external code from corrupting the object’s state

Public Methods in the BankAccount Class

```
class BankAccount:
    def __init__(self, initial_balance):
        if initial_balance < 0:
            raise ValueError("Initial balance cannot be <0")
        self.__balance = initial_balance

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount < 0:
            raise ValueError("Deposit amount cannot be <0")
        self.__balance += amount

    def withdraw(self, amount):
        if amount < 0:
            raise ValueError("Withdrawal amount cannot be <0")
        if self.__balance < amount:
            raise ValueError("Insufficient funds")
        self.__balance -= amount
```

Public Methods in the BankAccount Class

- The ‘deposit()‘ and ‘withdraw()‘ methods in ‘BankAccount‘ are public
- They allow external code to modify the object’s state in a controlled manner
- They perform validation to ensure that the object remains in a valid state
- External code cannot access the ‘balance‘ attribute directly, so it cannot corrupt the object’s state

Operator overloading

Operator Overloading in Object-Oriented Programming

Operator overloading is a technique in object-oriented programming that allows us to define the behavior of operators (+, -, *, /, etc.) for user-defined objects.

In Python, operator overloading is achieved by defining special methods (with double underscores) in a class that correspond to the operator being used.

For example, the addition operator (+) can be overloaded by defining the '__add__' method in a class.

Consider the following example:

Operator Overloading in Object-Oriented Programming

Suppose we have a class called ‘Vector‘ that represents a mathematical vector in 2D space. We can overload the addition operator (+) to allow adding two vectors together.

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

Here, the ‘__add__‘ method takes another ‘Vector‘ object as an argument, adds the corresponding components of the two vectors, and returns a new ‘Vector‘ object that represents the sum of the two vectors.

Using Overloaded Operators

Let's create two ‘Vector‘ objects and add them together using the overloaded addition operator (+):

```
v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3.x, v3.y)
```

Output: 4 6

Here, we create two ‘Vector‘ objects, ‘v1‘ and ‘v2‘, and add them together using the overloaded addition operator ‘+‘. The resulting ‘Vector‘ object, ‘v3‘, has components ‘(4, 6)‘.

Operator Overloading in Object-Oriented Programming

We can also overload other operators, such as the multiplication operator (*), to perform scalar multiplication of a vector.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

By overloading operators, we can make our code more intuitive and expressive, and allow our user-defined objects to behave more like built-in types.

List of Overloaded Operators in Python

Operator	Method Name	Description
+	`__add__`	Addition
-	`__sub__`	Subtraction
*	`__mul__`	Multiplication
/	`__truediv__`	Division
//	`__floordiv__`	Floor Division
%	`__mod__`	Modulo
**	`__pow__`	Exponentiation
<	`__lt__`	Less Than
<=	`__le__`	Less Than or Equal To
==	`__eq__`	Equal To
!=	`__ne__`	Not Equal To
>	`__gt__`	Greater Than
>=	`__ge__`	Greater Than or Equal To