Lecture 19 | OOP – Part II

Methods, state and controlled operations

Where we left off

- Every `BankAccount` object has its own owner and balance attributes
- The initializer assigns their starting values
- The `.deposit()` method changes the balance of the object on which it is called
- Now we will examine two questions: what does a method do to the object, and what does it return to the caller?

```
class BankAccount:

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

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

An object has a state

- The ***state*** of an object consists of the current values of its attributes
- For an account with `owner="Anna"` and `balance=100`, those attribute values describe its state at that moment
- A method may *inspect* the state, *modify* it, or both
- Different instances may have different states even if they belong to the same class (type)

Methods that read versus methods that change

- A method can **compute a result** from an attribute *without changing that attribute*
- Another method may **update** an attribute to store a new state
- These are different behaviors: printing or returning a value is not the same as modifying the object
- We will use `.get_balance()` and `.deposit()` to compare them

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

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

Returning a value from a method

- The return statement works as it does in any Python function
- `account.get_balance()` calls the method and produces the current balance as a result
- The result can be assigned to another variable or printed
- Reading the balance **does not modify** the stored attribute

```
# inside the class...
def get_balance(self):
    return self.balance

# Main code
a = BankAccount("Anna", 100)
result = a.get_balance()
print(result)     # 100
print(a.balance)  # 100
```

Changing state does not imply returning it

- `.deposit()` modifies `self.balance` but has no explicit return statement
- As with ordinary functions, a method without return produces None
- To inspect the updated balance, call `.get_balance()` or read the attribute
- Do not confuse an object's new state with the method's return value

```
a = BankAccount("Anna", 100)
result = a.deposit(40)

print(result)           # None
print(a.get_balance())  # 140
```

Why should a method check its inputs?

- Our first `.deposit()` version accepts any number, including negative amounts
- But **a negative deposit** would decrease the balance; that is not the operation we intend to model
- A useful class can **centralize rules that determine which operations are allowed**
- We already know the required programming tools: if statements, Boolean expressions, comparisons and return

Validation in .deposit()

- The method accepts an amount only when it is **strictly positive**
- If it is positive, the object's balance is increased and `True` is returned
- Otherwise, the balance remains unchanged and the method returns `False`
- The Boolean result tells the caller whether the requested operation **succeeded**

```
# inside the class...
def deposit(self, amount):
    if amount > 0:
        self.balance += amount
        return True

    return False
```

Tracing a rejected deposit

- Start with an account holding 100 EUR
- Calling `.deposit(-20)` makes the condition `amount > 0` False
- The assignment to `self.balance` is skipped, so the balance stays 100
- The method returns `False` to report the rejected operation

```
a = BankAccount("Anna", 100)
accepted = a.deposit(-20)

print(accepted)          # False
print(a.get_balance()) # 100
```

Adding .withdraw(): another state change

- A withdrawal should decrease the balance only for a positive amount not exceeding the available balance
- The `if` condition combines both requirements with `and`
- A successful withdrawal updates the attribute and returns `True`
- An invalid request returns `False` and leaves the balance unchanged

```
def withdraw(self, amount):
    if amount > 0 and amount <= self.balance:
        self.balance -= amount
        return True

    return False
```

Tracing successful and unsuccessful withdrawals

- Anna starts with 100 EUR
- Withdrawing 30 EUR succeeds; the balance becomes 70
- Then, withdrawing 90 EUR fails because 90 is greater than 70
- The two return values tell us which requests succeeded

```
a = BankAccount("Anna", 100)

print(a.withdraw(30))  # True

print(a.get_balance()) # 70

print(a.withdraw(90))  # False

print(a.get_balance()) # 70
```

Encapsulation: the practical idea

- ***Encapsulation*** means keeping the data (attributes) of an object together with the operations (methods) that manage it; it restricts direct access to internal data, allowing interaction only through a controlled interface.
- A deposit or withdrawal method can check the rules before changing the balance
- Writing `account.balance = -500` directly would bypass those checks; users of the class should normally prefer its intended methods
- Python does not automatically prohibit access to attributes merely because they belong to an object; we will not introduce advanced access-control mechanisms

Access Control and Encapsulation
Unlike languages such as Java and C++, which provide access modifiers such as `public`, `private`, and `protected`, Python mainly relies on **naming conventions**:
`_attribute`: indicates an internal attribute that should not be accessed directly from outside the class
`__attribute`: triggers *name mangling* (automatic renaming, e.g., `__attribute` becomes `_ClassName__value`), reducing accidental access but not enforcing strict privacy.making accidental access more difficult, but not preventing it
Python does not enforce strict privacy: both conventions can be bypassed, emphasizing programmer responsibility rather than access restrictions.

A complete BankAccount with controlled methods

- The class combines initialization, a read-only operation and two controlled state changes
- Every instance uses the same rules but stores its own independent balance
- The class is still written using only functions, conditions, arithmetic and dot notation
- The example on the right is a complete class definition

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def get_balance(self):
        return self.balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            return True
        return False

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            return True
        return False
```

Using two accounts independently

- Create two instances of `BankAccount`
- Each method call acts on the instance before the dot
- Changing Anna's balance does not alter Luca's balance
- We can inspect both accounts through the same `.get_balance()` method

```
a = BankAccount("Anna", 100)
b = BankAccount("Luca", 250)

a.deposit(30)
b.withdraw(50)

print(a.get_balance())  # 130
print(b.get_balance())  # 200
```

Using several objects in one program

- Objects are values that can be stored in lists, just like numbers, strings or dictionaries
- Each element of a list may refer to a different instance of the same class
- A for loop can visit each object and access its attributes or call its methods
- This lets us apply the same operation to many entities without creating separate variable names for every one

A list of BankAccount objects

- The expressions `BankAccount(...)` create three distinct accounts
- The list accounts stores references to the three objects in a chosen order
- During each iteration, account refers to the next object in the list
- Calling `.get_balance()` on each object reads its own state

```
accounts = [
    BankAccount('Anna', 100),
    BankAccount('Luca', 250),
    BankAccount('Mia', 150)
]

for account in accounts:
    print(account.owner, account.get_balance())
```

Output:

```
Anna 100
Luca 250
Mia 150
```

Calculating a total across objects

- We can combine familiar loops and accumulators with object methods
- Start with `total = 0`, then inspect every account in the list
- Calling `.get_balance()` produces a number; add that number to `total`
- The resulting total is computed from the actual object states, not from a separate copied list of values

```
total = 0
for account in accounts:
    total += account.get_balance()

print('Total balance:', total)
```

Output:

```
Total balance: 500
```

A method can raise an exception

- A method can reject an invalid operation by raising an exception with raise
- Here, `.withdraw()` checks the requested amount before changing the balance
- `ValueError` signals that the supplied value violates the method's rules
- Once `raise` executes, the method stops and the balance remains unchanged

```
class BankAccount:

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def withdraw(self, amount):

        if amount <= 0:
            raise ValueError(
                'Amount must be positive')

        if amount > self.balance:
            raise ValueError(
                'Insufficient funds')

        self.balance -= amount
```

Handling an exception outside the object

- The withdrawal amount comes from the user, so it may violate the account rules
- The try block calls `.withdraw()`, which may raise `ValueError`
- The `except` block receives the error and displays a message without ending the program
- Assume the user enters a valid numeric value; input conversion errors are not considered here

```
account_alice = BankAccount('Alice', 100)

amount = float(input('Withdraw amount: '))
try:
    account_alice.withdraw(amount)
    print('Withdrawal completed')
except ValueError as error:
    print('Withdrawal failed:', error)

print('Balance:', account_alice.balance)
```

Example input: -20

Output:

```
Withdrawal failed: Amount must be positive
Balance: 100
```

A method can handle an error internally

- A method can contain its own try-except when it knows how to handle a possible failure
- This method receives a text amount and tries to convert it to `float`
- If conversion fails, the operation is cancelled safely and `False` is returned
- Otherwise, `.deposit()` applies the usual rules and returns its result

```python
# Method added inside BankAccount
def deposit_from_text(self, text):

    try:
        amount = float(text)

    except ValueError:
        print('Invalid amount')
        return False

    return self.deposit(amount)
```

Internal vs external error handling

- The caller invokes `.deposit_from_text()` without having to manage conversion exceptions
- The object handles a recoverable problem by skipping the invalid deposit
- Contrast: `.withdraw()` raises an exception and delegates handling to the calling program
- An exception is useful when the caller must be informed of a failure and choose how to respond

```
# Using BankAccount with deposit() defined
account_alice = BankAccount('Alice', 100)

account_alice.deposit_from_text('50')
account_alice.deposit_from_text('abc')

print(account_alice.balance)
```

Output:

```
Invalid amount
150.0
```

What we can now do with objects

- An object’s state is represented by its attributes, which can change over time
- A method may return a value, modify the state, do both or do neither
- Validation inside methods allows the class to implement simple rules consistently
- Next we will integrate several objects with lists, loops and functions to build a complete small program