Lecture 20 | OOP – Part III

Inheritance, specialization and method overriding

Starting point: classes and separate instances

- We already know how a `BankAccount` stores its owner and balance and how its methods read or modify them
- We can create many accounts from this class, store them in a list and use a loop to work with their states
- What if we also need a **special kind of account** with **additional behavior**, while **keeping the ordinary account operations**?
- ***Inheritance*** offers a way to define such a specialized class without rewriting every shared method

The existing class: BankAccount

- `BankAccount` will be the base class in our example
- Its initializer stores the same two attributes in every account object
- The `.deposit()` method checks a positive amount and updates the balance; its Boolean return value reports success
- A derived class will later reuse this behavior

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

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

Inheritance: the basic idea

- A **base class** (also called *parent class*) defines attributes and methods common to several related kinds of objects
- A **derived class** (also called *child class*) inherits the available behavior of the base class and may introduce additional behavior
- A `SavingsAccount` is a particular kind of `BankAccount`: it still has an *owner*, a *balance* and the ability to *deposit* funds
- Inheritance models an “is-a” relationship and supports code reuse; it should not be used merely because two classes happen to share a few values

Defining a derived class: the syntax

- Sintax: When defining, write the name of the base class in parentheses after the new class name

```
class DerivedClass(BaseClass):
```

- The keyword pass creates an otherwise empty class body; it is useful here to show that inheritance already works, and fill it later.
- `SavingsAccount` **inherits the existing initializer** and `deposit` method from `BankAccount`
- No new attributes or methods are defined yet

```
class SavingsAccount(BankAccount):
    pass

s = SavingsAccount('Anna', 1000)
print(s.owner, s.balance)
```

Output:

```
Anna 1000
```

Calling an inherited method

- Although `.deposit()` is not written in the child class, its objects can still call it
- Python looks
    1. for the method on the class of the object
    2. and then among its base classes when necessary
- The inherited method operates on `s`, so `self` refers to the `SavingsAccount` instance
- The account remains a separate object with its own stored balance

```
s = SavingsAccount('Anna', 1000)
s.deposit(200)

print(s.balance)
```

Output:

```
1200
```

Adding a new method to the derived class

- `SavingsAccount` may define an additional operation that the general `BankAccount` does not offer
- The method `.add_interest()` uses the inherited balance attribute and updates it; multiplier `rate` is given to the method as a numerical argument.
- Other `BankAccount` instances do not gain this specialized method (`AttributeError` if you try to use it)
- The subclass now reuses existing methods and adds a new one

```
Python example
class SavingsAccount(BankAccount):
    def add_interest(self, rate):
        self.balance += self.balance * rate

s = SavingsAccount('Alice', 1000)
s.deposit(200)
s.add_interest(0.05)
print(s.balance)
```

Output: 1260.0

Why a derived class may need its own initializer

- Our savings accounts **should have a fixed interest rate stored in each object**, rather than requiring the rate as an argument every time
- That means a `SavingsAccount` now needs three attributes: `owner`, `balance` and `interest_rate`
- The inherited `__init__()` only accepts `owner` and `balance`, so we must define **a new initializer** in `SavingsAccount`
- The key question is how to initialize the two shared attributes *without duplicating the code* of `BankAccount`

What is super() and why do we use it?

- `super()` is a built-in function that gives access to inherited methods
- In simple single inheritance, `super().method()` invokes the parent-class implementation of the method; it calls the method and actually executes the operations described in the parent class
- Use it to **reuse existing behavior** instead of copying code into the child class
- The inherited method operates on the same instance, not on a new parent object

Compare the two initializer implementations

```python
# Repeating the parent initialization
def __init__(self, owner, balance, rate):
    self.owner = owner
    self.balance = balance
    self.interest_rate = rate

# Reusing the parent initialization
def __init__(self, owner, balance, rate):
    super().__init__(owner, balance)
    self.interest_rate = rate
```

Using super().__init__() to initialize a child object

- A child class without its own `__init__()` inherits the initializer from the parent
- When the child defines `__init__()`, the parent initializer is not called automatically
- `super().__init__(owner, balance)` explicitly reuses the parent initialization on this same object
- Do not pass `self`: the method accessed through `super()` is already bound to the current instance
- Then `self.interest_rate` stores the attribute specific to `SavingsAccount`

```
class SavingsAccount(BankAccount):
    def __init__(self, owner, balance, rate):
        super().__init__(owner, balance)
        self.interest_rate = rate

# Main code ----------------------------------
acc_alice = SavingsAccount('Alice', 1000, 0.05)
print(acc_alice.owner)          # Alice
print(acc_alice.balance)        # 1000
print(acc_alice.interest_rate)  # 0.05
```

Following initialization step by step

- `SavingsAccount("Anna", 1000, 0.05)` creates a new `SavingsAccount` object
- The child initializer receives `owner`, `balance` and `rate`; `self` refers to the newly created object
- `super().__init__(owner, balance)` sets `self.owner` and `self.balance` using the shared initializer
- Then `self.interest_rate = rate` adds the specialized attribute; no second object is created by `super()`

```
super().__init__(owner, balance)
self.interest_rate = rate
```

```
s = SavingsAccount('Alice', 1000, 0.05)

# State after initialization:
# s.owner = Alice
# s.balance = 1000
# s.interest_rate = 0.05
```

Using a stored attribute in a new method

- Now that the interest rate is stored on the instance, `.add_interest()` no longer needs a rate argument
- The method reads `self.balance` and `self.interest_rate`, computes the interest and updates the balance
- All operations act on the same savings account object
- This also illustrates how an instance attribute may be numeric, textual or another Python value

```
class SavingsAccount(BankAccount):
    def __init__(self, owner, balance, rate):
        super().__init__(owner, balance)
        self.interest_rate = rate

    def add_interest(self):
        self.balance += (self.balance *
                         self.interest_rate)

s = SavingsAccount('Alice', 1000, 0.05)
s.add_interest()
print(s.balance)  # 1050.0
```

Method overriding: changing inherited behavior

- Inheritance also allows the derived class to **define a method with the same name** as one in the base class
- When the method is called on a derived-class instance, the derived version is used: this is ***method overriding***
- Overriding is useful when the specialized class must **implement the operation differently**
- An override may **entirely replace** the original behavior or call it first using `super().method()` and then **add further steps**

An overriding example without super()

- The base class defines `.describe()` to return a general account description
- The derived class provides a different version of `.describe()` with the same name
- Calling `.describe()` on a normal `BankAccount` and on a `SavingsAccount` selects the appropriate implementation
- This is a simple example of **polymorphic behavior**: the same method call can produce different results

```
class BankAccount:
    def describe(self):
        return 'Ordinary account'

class SavingsAccount(BankAccount):
    def describe(self):
        return 'Savings account'

print(BankAccount().describe())
print(SavingsAccount().describe())
```

Output:

```
Ordinary account
Savings account
```

Using super() inside an overridden method

- **An override can first call the inherited implementation** and then add more information
- `super().describe()` invokes the method provided by the base class using the current instance
- This avoids rewriting the shared part of the method
- `super()` is useful both in `__init__()` and in other overridden methods

```
class BankAccount:
    def describe(self):
        return 'Account'

class SavingsAccount(BankAccount):
    def describe(self):
        base_text = super().describe()
        return base_text + ' (savings)'

print(SavingsAccount().describe())
```

Output:

```
Account (savings)
```

The complete savings-account example

- `SavingsAccount` reuses initialization and the existing deposit operation from `BankAccount`
- It introduces an `interest_rate` attribute with the help of `super().__init__()`
- Its new method `.add_interest()` changes the state according to that rate
- The output confirms that inherited and specialized methods operate on the same object

Base class – BankAccount

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

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

    def describe(self):
        return 'Account'
```

Derived – SavingsAccount

```
class SavingsAccount(BankAccount):
    def __init__(self, owner, balance, rate):
        super().__init__(owner, balance)
        self.interest_rate = rate

    def add_interest(self):
        self.balance += (self.balance *
                         self.interest_rate)

    def describe(self):
        return super().describe() + ' (savings)'
```

Main program

```
savings_alice = SavingsAccount('Alice', 1000, 0.05)
savings_alice.deposit(200)
savings_alice.add_interest()
print(savings_alice.describe())  # Account (savings)
print(savings_alice.balance)     # 1260.0
```

The complete savings-account example

- `SavingsAccount` reuses initialization and the existing deposit operation from `BankAccount`
- It introduces an `interest_rate` attribute with the help of `super().__init__()`
- Its new method `.add_interest()` changes the state according to that rate
- The output confirms that inherited and specialized methods operate on the same object

Base class – BankAccount

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

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

    def describe(self):
        return 'Account'
```

Derived – SavingsAccount

```
class SavingsAccount(BankAccount):
    def __init__(self, owner, balance, rate):
        super().__init__(owner, balance)
        self.interest_rate = rate

    def add_interest(self):
        self.balance += (self.balance *
                         self.interest_rate)

    def describe(self):
        return super().describe() + ' (savings)'
```

NOTE: Related classes may be definend in the same module or separate; in this case, the child must `import` the parent in order to use its name. Same for the main

Main program

```
savings_alice = SavingsAccount('Alice', 1000, 0.05)
savings_alice.deposit(200)
savings_alice.add_interest()
print(savings_alice.describe())  # Account (savings)
print(savings_alice.balance)     # 1260.0
```

Reading inheritance code: common mistakes

- Inheritance syntax uses parentheses in the class definition: `class SavingsAccount(BankAccount):`
- If the derived class does not define `__init__()`, the parent initializer is inherited; if it defines its own, the parent initializer is not called automatically
- Use `super().__init__(...)` when the parent initialization should also happen, with arguments matching the parent signature
- The name `self` is written in instance-method definitions but is passed automatically in calls such as `s.deposit(200)`
- An overridden method has the same name as the original; a new method has a new name

Further Reading

Operator Overloading

Operators and Python special methods

How Python evaluates an operator

Further Reading

- We already use + for numbers (3 + 4) and for strings ("Py" + "thon")
- Python supports these operations through special methods defined by the corresponding classes
- For example, `a + b` normally starts by calling `a.__add__(b)`: the **method belongs to the left object** a, while b is **passed as an argument**
- Python calls this method automatically when we write +; we do not need to call __add__() ourselves
- If the operation is not supported by either operand, Python raises `TypeError`

The special methods behind familiar operations

Further Reading

Expression	Special method
`a + b`	`a.__add__(b)`
`a - b`	`a.__sub__(b)`
`a * b`	`a.__mul__(b)`
`a / b`	`a.__truediv__(b)`
`a % b`	`a.__mod__(b)`
`a == b`	`a.__eq__(b)`
`a < b`	`a.__lt__(b)`
`a <= b`	`a.__le__(b)`
`a > b`	`a.__gt__(b)`
`a >= b`	`a.__ge__(b)`
`str(a)`	`a.__str__()`

- These names are fixed and recognized by Python
- An operator automatically invokes the corresponding special method, when supported by the class
- Custom classes may define these methods to specify their own behavior
- .__str__() is called for readable text representation, rather than an operator

We can define these methods in our own classes

Further Reading

- Special methods are defined inside a class, just like other methods
- Their names are fixed: Python expects `__add__()` for the + operator
- Let's consider a class `Vector2D` that stores two coordinates: x and y
- Its initializer creates a vector, but + is not yet defined for two Vector2D objects

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

v1 = Vector2D(2, 3)
v2 = Vector2D(4, 1)

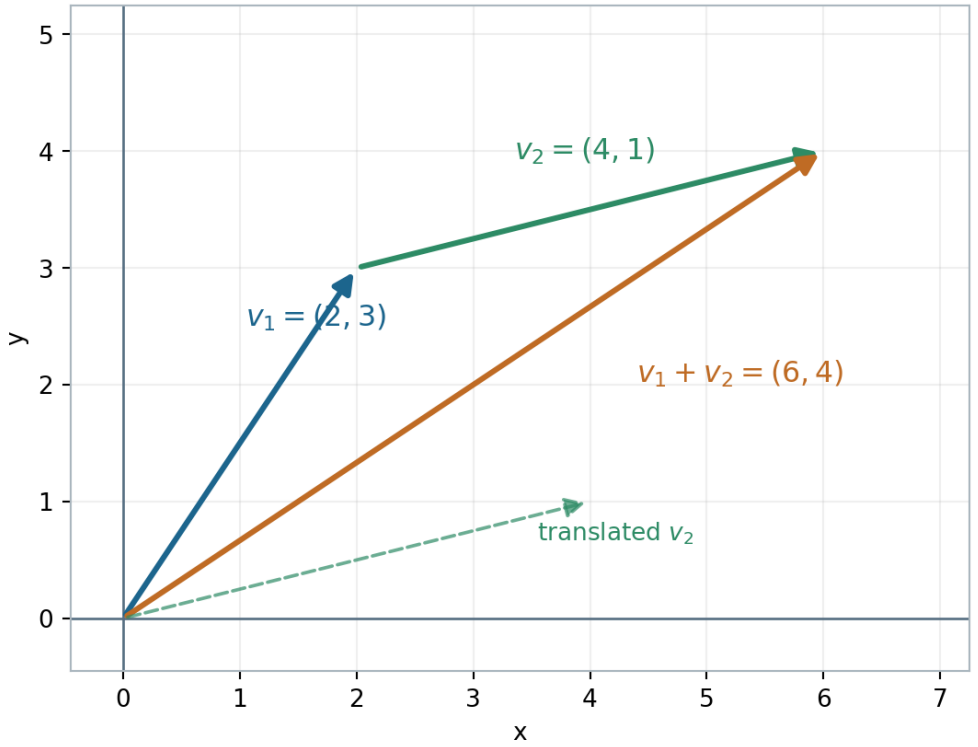
# v1 + v2 is not supported yet
```

We will give + a meaning for Vector2D

First, what does adding two 2D vectors mean?

Further Reading

- A vector has an x component and a y component
- To add two vectors, means to add their corresponding components
- The result is a new vector, without changing either original vector
- We want to write this naturally as v1 + v2 in Python



ISEG Lisbon School of Economics & Management Universidade de Lisboa

Implementing vector addition with __add__()

Further Reading

- For `v1 + v2`, Python first attempts `v1.__add__(v2)`
- Inside `__add__(self, other)`, self is `v1` and other is `v2`
- Add v1.x to v2.x and v1.y to v2.y, returning a new `Vector2D`
- The original vectors stay unchanged, so sums can be chained: v1 + v2 + v3

```
# Inside class Vector2D
def __add__(self, other):
    x_sum = self.x + other.x
    y_sum = self.y + other.y
    return Vector2D(x_sum, y_sum)

v3 = v1 + v2
print(v3.x, v3.y)  # 6 4
```

Why might we also define __str__()?

Further Reading

- With `__add__()`, `v1 + v2` produces a `Vector2D` object
- Printing a custom object may otherwise show its type and memory address, rather than its coordinates
- Python automatically uses `__str__()` for readable string conversion, including in `print()`
- Our `__str__()` returns the coordinates as `(x, y)`

```
# Inside class Vector2D
def __str__(self):
    return f'({self.x}, {self.y})'

print(v1)          # (2, 3)
print(v2)          # (4, 1)
print(v1 + v2)     # (6, 4)
```

The complete Vector2D example

Further Reading

- `__init__()` stores the vector components
- `__add__()` defines the + operation and returns a new `Vector2D`
- For `v1 + v2`, Python automatically starts with `v1.__add__(v2)`
- `__str__()` determines how `print()` displays the vector
- We define these methods inside the class, but use operators and `print()` to invoke them

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector2D(
            self.x + other.x,
            self.y + other.y)
    def __str__(self):
        return f'({self.x}, {self.y})'
v1 = Vector2D(2, 3)
v2 = Vector2D(4, 1)
print(v1 + v2)  # (6, 4)
```

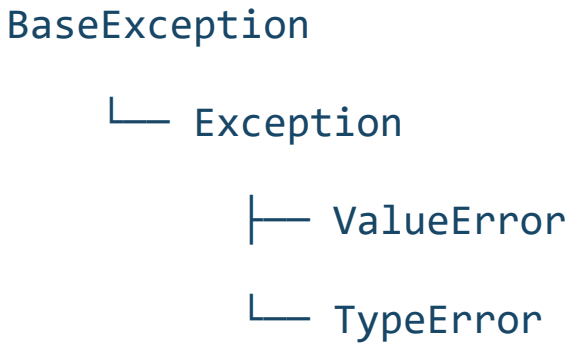
Exceptions as Classes

Also errors are classes and have a hierarchy

Exceptions are classes too

- We already know that `raise` signals an error and try-except can handle it
- Built-in exceptions, such as `ValueError` and `TypeError`, **are also Python classes**
- They **belong to a hierarchy**: a more specific exception inherits from a more general class
- This means the inheritance rules from this lecture also apply to errors

Simplified hierarchy



`ValueError` is a kind of **`Exception`**

Defining a custom exception class

- Suppose a bank account needs to report a specific problem: a withdrawal exceeds the balance
- We can give that problem its own exception type instead of using a generic `ValueError`
- The new class inherits the normal behavior of Python exceptions
- For this simple case, no custom `__init__()` or additional methods are needed

```
class InsufficientFundsError(ValueError):
    pass

# A specific type of ValueError
# The name identifies the problem

# some operations...
raise InsufficientFundsError(
    "Not enough funds"
)
```

Raising a custom exception in a method

- We reuse `BankAccount` and change only how `.withdraw()` reports insufficient funds
- First check that the **amount is positive**, then check that the **balance is sufficient**
- If a check fails, raise an appropriate error before changing the balance
- The balance changes only when all checks have succeeded

```
class InsufficientFundsError(ValueError):
    pass

# ------------------------------------------------

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

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Positive amount required")
        if amount > self.balance:
            raise InsufficientFundsError("Not enough funds")
        self.balance -= amount
```

Catching the error outside the object

- The caller chooses the withdrawal amount: the method cannot assume it is acceptable
- The call is inside `try` because `.withdraw()` may raise `InsufficientFundsError`
- If it raises, Python skips the success message and executes the matching except block
- The object remains valid: the withdrawal did not change its balance
- For simplicity, assume the user enters a numeric value

```
account_alice = BankAccount("Alice", 100)
amount = float(input("Withdraw amount: "))

try:
    account_alice.withdraw(amount)
    print("Withdrawal completed")
except InsufficientFundsError as error:
    print("Withdrawal failed:", error)

print(account_alice.balance)
```

Output if the input is 150:

```
Withdrawal failed: Not enough funds
100
```

A parent exception can catch a child exception

- `InsufficientFundsError` inherits from `ValueError`
- Therefore, `except ValueError` also matches `InsufficientFundsError` (`InsufficientFundsError` ***is*** also a `ValueError`)
- Use **the more specific except first** when different problems need different responses
- Python executes only the first matching except block
- → if you reverse the order (generic then specific), the most generic would intercept also the specific, and the second condition is never reached

```
try:
    account_alice.withdraw(amount)
except InsufficientFundsError:
    print("Not enough balance")
except ValueError:
    print("Amount must be positive")
```

Input 150 → `Not enough balance`
Input -10 → `Amount must be positive`

Reversing these `except` blocks would hide the more specific case

When should we create custom exceptions?

- Use a built-in exception when its meaning is already clear, such as `ValueError` for a nonpositive amount
- Create a custom exception when the program benefits from distinguishing a particular problem
- An exception can be raised inside an object method and handled by the caller
- Inheritance lets the caller catch either the specific error or its parent type
- Custom exceptions are ordinary classes; we can keep them simple

Key patterns

```
class MyError(ValueError):
    pass

# In a method/function
raise MyError("Problem details")

# In the calling program
try:
    obj.method()
except MyError as error:
    print(error)
```

Inheritance: quick reference

- **Base class** and **derived class**: define a general type and a specialized type using class `Child(Parent):`
- **Inherited method**: a child instance can call a method defined in the parent without rewriting it
- **New method**: the child adds behavior specific to its own objects
- **Overriding**: the child provides a different implementation of an inherited method
- `super()`: inside the child, reuse the parent behavior, especially initialization with super().__init__(...)
- Exam-level applications use one base class, one simple derived class and short method calls; no multiple inheritance is required