High-Performance Computing for Cyber-Physical Systems

Lingjie Zhang

2024 SS

This course is based on the lecture ED160021 of Technical University of Munich

This note is a record of some messy knowledge points, which may not be in order, but most of them are parts of my first contact or learning.

Contents

1 Best Practice for Resource Cleanup **1**

1.1 Context Managers 1

1.2 Explicit Cleanup Methods 1

2 Mixin **1**

3 Exception classes **2**

4 Encapsulation **3**

4.1 Private and Public Attributes 3

5 Properties and the @property Decorator **4**

6 Abstract Classes and Protocols **5**

7 Pythonic Design Patterns **7**

7.1 Singleton Pattern 7

7.2 Factory Pattern 7

7.3 Observer Pattern 8

7.4 Decorator Pattern 9

7.5 The Strategy Pattern 11

1 Best Practice for Resource Cleanup

1.1 Context Managers

Use the `with` statement to create a context manager that automatically handles resource cleanup when the block is exited. This is the recommended approach for file handling, network connections, and other resources that support context managers. For example:

```
with open("file.txt", "r") as file:
    content = file.read()
# file is automatically closed when the block is exited
```

1.2 Explicit Cleanup Methods

Some resources may not support context managers, or you may need more control over when the resource is cleaned up. In such cases, use explicit cleanup methods provided by the resource, and make sure to call them when the resource is no longer needed. For example:

```
connection = SomeDatabaseConnection()
try:
    # Perform database operations
    pass
finally:
    connection.close()  # Clean up the connection explicitly
```

2 Mixin

A mixin is a special kind of superclass that is designed to provide specific functionality to other classes through multiple inheritance, without being considered a complete class on its own. Mixins are usually small and focused, implementing a single piece of functionality that can be easily combined with other classes. By using mixins, you can create modular and reusable components that can be easily added to other classes as needed.

A example of using a mixin:

```
class SpeakMixin:
    def speak(self):
        print("Hello!")

class Person:
    pass
```

```
class Employee(Person, SpeakMixin):
    pass

e = Employee()
e.speak() # Hello!
```

3 Exception classes

Some common built-in exception classes in Python:

- **SyntaxError**: Indicates a syntax error, typically occurring during the compilation of code when it doesn't conform to Python's syntax rules.
- **IndentationError**: Denotes an indentation error, usually occurring when there's incorrect indentation within a code block.
- **NameError**: Occurs when trying to reference a variable or function name that hasn't been defined.
- **TypeError**: Indicates an error caused by an operation being applied to an object of an inappropriate or incompatible type.
- **ValueError**: Indicates an error caused by a value being inappropriate for the operation it's used for.
- **AttributeError**: Occurs when attempting to access an attribute that doesn't exist for an object.
- **KeyError**: Indicates an error caused by attempting to access a key that doesn't exist in a dictionary.
- **IndexError**: Occurs when trying to access an index that doesn't exist in a sequence.
- **FileNotFoundError**: Indicates that a file being accessed was not found.

- **IOError**: Denotes an I/O error, typically occurring when an input/output operation fails or is interrupted.

- **ZeroDivisionError**: Indicates an error caused by attempting to divide by zero.

- **OverflowError**: Occurs when a numeric computation result exceeds the representation limit.

- **RuntimeError**: Denotes a runtime error, typically occurring when an unrecoverable error condition is encountered during execution.

- **StopIteration**: Indicates that an iteration has stopped, often raised when an iterator's `next()` method has no more elements to return.

4 Encapsulation

4.1 Private and Public Attributes

- **Public attributes**: These attributes are accessible from any part of the code and are intended for external use. By default, all attributes in a Python class are public. There is no special naming convention for public attributes

- **Private attributes**: These attributes are meant to be used only within the class and are not intended for external access. To indicate that an attribute is private, you should prefix its name with a double underscore (e.g., `__private_attribute`).

Here's an example that demonstrates the use of public and private attributes:

```
class BankAccount :
    def __init__ ( self , account_number , initial_balance ) :
        self . account_number = account_number
        self . __balance = initial_balance

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

    def withdraw ( self , amount ) :
        if amount <= self . __balance :
            self . __balance -= amount
        else :
            print (" Insufficient funds ")
```

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

    account = BankAccount (12345 , 1000)
    print ( account . account_number ) # 12345
    print ( account . get_balance () ) # 1000
    print ( account . __balance ) # AttributeError : 'BankAccount'
                                        object has no attribute '
                                        __balance '
```

In this example, the `BankAccount` class has a public attribute `account_number` and a private attribute `__balance`. The private attribute can only be accessed and modified within the class, through methods like `deposit`, `withdraw`, and `get_balance`. Trying to access the private attribute directly from outside the class would result in an `AttributeError`.

5 Properties and the @property Decorator

The `@property` decorator is used to define a method as a property. When a property is accessed like an attribute, the method decorated with `@property` is called. This allows us to implement controlled access to an attribute without the need for explicit getter methods.

To create a setter for a property, we can use the `@attribute.setter` decorator. This decorator defines a method as the setter for a property, which is called when the property is assigned a new value. It provides a clean way to implement validation or other logic when modifying an attribute

Let's rewrite the previous `BankAccount` example using properties:

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

    @property
    def balance(self):
        return self._balance

    @balance.setter
    def balance(self, new_balance):
        if new_balance < 0:
            raise ValueError("Balance cannot be negative.")
        self._balance = new_balance

account = BankAccount(100)
print(account.balance) # 100
account.balance = 200
print(account.balance) # 200
```

6 Abstract Classes and Protocols

Abstract classes in Python are classes that define a common interface for their subclasses, but cannot be instantiated themselves.

To define an abstract class in Python, you can use the abc module, which provides the ABC (Abstract Base Class) metaclass. By inheriting from this metaclass, you can create your own abstract classes. Here's an example of defining an abstract class:

```
from abc import ABC

class Shape(ABC):
    pass
```

In this example, we define an abstract class Shape by inheriting from the ABC metaclass provided by the abc module. This class can serve as a base class for various shapes, but it cannot be instantiated directly:

```
shape = Shape() # TypeError : Can't instantiate abstract class
                                Shape with abstract methods
                                area
```

Abstract Methods and the @abstractmethod Decorator

Abstract methods are a way to enforce a common interface for subclasses, ensuring that they provide implementations for all required methods. In Python, you can define abstract methods using the @abstractmethod decorator, which is also provided by the `abc` module.

Here's an example:

```
from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod
    def area(self):
        pass


class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2
```

If a subclass does not provide an implementation for an abstract method, it will also be considered an abstract class, and attempting to instantiate it will result in a `TypeError`:

```
class Square(Shapre):
    pass

square = Square() # TypeError: Can't instantiate abstract class
                                    Square with abstract methods
                                    area
```

Implementing Protocols and Duck Typing "Duck Typing" means that you can use an object in a specific context as long as it implements the necessary methods or properties, regardless of its actual class. This approach provides great flexibility and allows for more generic and reusable code.

To define a protocol in Python 3.8 or later, you can use the `typing.Protocol` class:

```
from typing import Protocol

class IterableProtocol(Protocol):
    def iter(self):
pass
```

In this example, the `IterableProtocol` defines a single method, `iter`, which means any class implementing this method would conform to the protocol. The function using this protocol might look like this:

```
def sum_elements(iterable: IterableProtocol):
    total = 0
    for element in iterable:
        total += element
    return total
```

This function uses duck typing and the `IterableProtocol`, so it doesn't care about the specific type of the input iterable. It only requires that the input implements the iteration protocol. This means that the function can work with lists, tuples, sets, and even custom iterable classes:

```
print(sum_elements([1, 2, 3])) # 6
print(sum_elements((1, 2, 3))) # 6
print(sum_elements({1, 2, 3})) # 6

class CustomIterable:
    def iter(self):
        return iter([1, 2, 3])

print(sum_elements(CustomIterable())) # 6
```

7 Pythonic Design Patterns

7.1 Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is useful when you want to manage a shared resource or enforce a single point of control over a particular part of your application.

Here's one way to implement the Singleton pattern in Python using a class-level attribute:

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance

singleton1 = Singleton()
singleton2 = Singleton()

print(singleton1 is singleton2) # True
```

7.2 Factory Pattern

The Factory pattern is a creational pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern is useful when you want to create objects of different classes, sharing a common interface or base class, without specifying the exact class to be created.

Here's an example of how to implement the Factory pattern in Python:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

```
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "Dog":
            return Dog()
        elif animal_type == "Cat":
            return Cat()
        else:
            raise ValueError("Invalid animal type")

animal1 = AnimalFactory.create_animal("Dog")
animal2 = AnimalFactory.create_animal("Cat")

print(animal1.speak()) # Woof!
print(animal2.speak()) # Meow!
```

7.3 Observer Pattern

The Observer pattern is a behavioral design pattern that establishes a one-to-many dependency between objects, where one object (the subject) maintains a list of its dependents (observers) and notifies them automatically of any changes in state. This pattern is useful when there is a need to maintain consistency between related objects or to update multiple objects when a single object changes.

Consider a scenario where we have a data source and multiple components that display or process the data in different ways. Whenever the data source changes, these components need to be updated accordingly. The Observer pattern can help us achieve this without tightly coupling the data source and the components.

Here's an example implementation of the Observer pattern in Python:

```
class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self):
        for observer in self._observers:
            observer.update(self)

class DataSource(Subject):
    def __init__(self, value):
        super().__init__()
        self._value = value
```

```
        @property
        def value(self):
            return self._value

        @value.setter
        def value(self, new_value):
            self._value = new_value
            self.notify()

    class DisplayObserver:
        def update(self, subject):
            print(f"Display: The new value is {subject.value}")

    class LoggerObserver:
        def update(self, subject):
            print(f"logger: Recorded new value: {subject.value}")

    data_source = DataSource(42)
    display = DisplayObserver()
    logger = LoggerObserver()

    data_source.attach(display)
    data_source.attach(logger)

    data_source.value = 10
```

7.4 Decorator Pattern

Decorators are a powerful and flexible pattern in Python that allows you to modify or enhance the behavior of functions or classes without altering their code. They are widely used in Python for a variety of purposes, such as enforcing access control, logging, memoization, and more.

Functions Decorating Functions

Function decorators are a convenient way to wrap a function with additional functionality. They are typically used when you have a simple use case that doesn't require maintaining state or when the decorator logic is straightforward.

Here's an example of a function decorator:

```
    def function_decorator(func):
        def wrapper(*args, **kwargs):
            print("Before calling the original function.")
            result = func(*args, **kwargs)
            print("After calling the original function.")
            return result
        return wrapper
```

```
@function_decorator
def my_function():
    print("Inside the original function.")

my_function()
```

Functions Decorating Classes

Sometimes you may want to apply a decorator to a class instead of a function. This can be useful when you need to modify or enhance the behavior of an entire class, such as adding or modifying class methods or attributes.

Functions can be used to decorate classes, as shown in the following example:

```
def class_decorator(cls):
    original_method = cls.original_method

    def decorated_method(self):
        print("Before calling the original method.")
        original_method(self)
        print("After calling the original method.")

    cls.decorated_method = decorated_method
    return cls

@class_decorator
class MyClass:
    def original_method(self):
        print("Inside the original method.")

my_instance = MyClass()
my_instance.decorated_method()
```

Classes Decorating Classes

When the decorator logic becomes more complex or needs to maintain some state, using a class decorator can provide greater flexibility and reusability. Class decorators can be extended and customized through inheritance and can store state as instance variables.

Here's an example of a class decorator:

```
class ClassDecorator:
    def init(self, cls):
        self._cls = cls

    def __call__(self, *args, **kwargs):
        wrapped_instance = self._cls(*args, **kwargs)
        wrapped_instance.decorated_method = self._decorated_method
```

```
        return wrapped_instance

    def _decorated_method(self):
        print("Before calling the original method.")
        self.original_method()
        print("After calling the original method.")

    @ClassDecorator
    class MyClass:
        def original_method(self):
            print("Inside the original method.")

    my_instance = MyClass()
    my_instance.decorated_method()
```

The @ symbol

It's important to note that the @ symbol used in decorators is merely syntactic sugar that simplifies the process of applying decorators to functions or classes. Using @decorator before a function or class definition is equivalent to calling the decorator function with the original function or class as an argument and then assigning the result back to the original name.

For example, the following two code snippets are functionally equivalent:

```
@decorator
def my_function():
    pass

# is equivalent to

def my_function():
    pass
my_function = decorator(my_function)
```

7.5 The Strategy Pattern

The Strategy Pattern is a behavioral design pattern that allows you to define a family of algorithms, encapsulate each of them in a separate class, and make them interchangeable at runtime. By using the Strategy Pattern, you can alter the behavior of an object without modifying its code or structure. This pattern is particularly useful when you need to provide multiple ways to perform an operation and want to be able to switch between them easily.

Here's an example that demonstrates the Strategy Pattern in Python:

```
from abc import ABC, abstractmethod
```

```
class SortingStrategy(ABC):
    @abstractmethod
    def sort(self, data):
        pass

class QuickSort(SortingStrategy):
    def sort(self, data):
        # Implementation of the QuickSort algorithm
        pass

class MergeSort(SortingStrategy):
    def sort(self, data):
        # Implementation of the MergeSort algorithm
        pass

class Context:
    def init(self, strategy: SortingStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: SortingStrategy):
        self._strategy = strategy

    def execute_sort(self, data):
        self._strategy.sort(data)

# Usage example

data = [5, 3, 1, 4, 2]

context = Context(QuickSort())
context.execute_sort(data)

context.set_strategy(MergeSort())
context.execute_sort(data)
```