Scientific Software Development with Python

Design patterns

Simon Pfreundschuh
Department of Space, Earth and Environment

CHALMERS
UNIVERSITY OF TECHNOLOGY

	Conceptual	Technical
Organisational	Project planning & management	Version control, testing, deployment (DevOps)
Implementational	Software design	Python programming, scientific computing

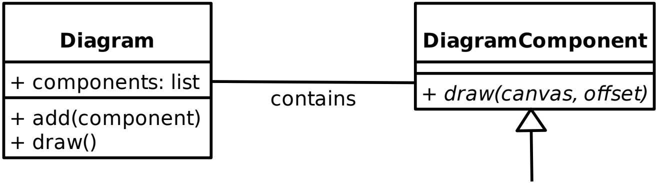
PART 2

This lecture

- Object oriented design patterns:
  - General description
  - Implementation in Python

Object oriented design

- Modeling the real world using classes and their relations to each other
- Keep it dry, keep it shy:
  - Every class should have a single, unique responsibility
  - Classes are decoupled from each other by defining interface
- Example: The `DiagramComponent` interface from last lecture



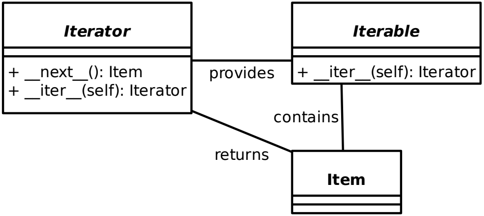
Design patterns

- Generalized, object-oriented solution for common design problems
- Motivation: Create a common language to solve reoccurring problems in software design

Iterators and iterables

- An **iterator** is an object that implements a loop over a sequence of objects
- An **iterable** is an object that provides access to a sequence of objects to iterate over

UML Diagram



`Iterable`

- Interface for container objects that provide access to a sequence of objects.
- ABC (standard library): `collectaion.abc.Iterable`
- Required class method: `__iter__`
    - Should return *iterator object*.

`Iterator`

- General protocol for *iterators* that implement the looping over object in a collection.
- ABC (standard library): `collections.abc.Iterator`
- Required class methods:
  - `__next__`: Should return next object in collection or `StopIteration` when exhausted
  - `__iter__`: Should return iterator object itself

Iterating over an iterator (the verbose way)

- The iterator protocol defines a generic way to loop over the elements of a container:

```python
iterator = iterable.__iter__()
while True:
    try:
        item = iterator.__next__()
        item.do_something()
    except StopIteration:
        break
```

- Instead of the special member funcs `__iter__` and `__next__`, it is also possible to use `iter(iterable)` and `next(iterator)` builtin functions.

Iterating over an iterator (the Pythonic way)

- The iterator pattern is so ubiquitous that it has *language-level support* in Python:

```
for item in iterable:
    item.do_something()
```

By implementing the **iterator protocol** you can use your own classes in Python for loops.

From iterators to generators

- The iterator pattern hides away the details of element storage from the user that consumes its elements.
- In some cases it can be desirable to *not store the elements at all*:

```python
class Squares:
    def __init__(self, start, stop):
        self.value = start
        self.stop = stop

    def __iter__(self):
        return self

    def __next__(self):
        if self.value >= self.stop:
            raise StopIteration()
        square = self.value ** 2
        self.value += 1
        return square
```

Generator functions

- Python provides special syntax to simplify the implementation of generators:

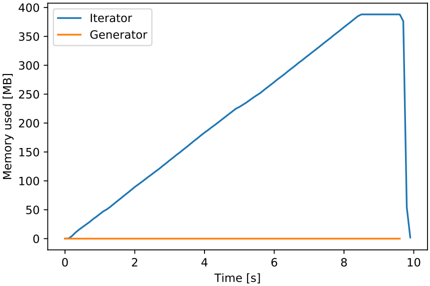
```
def squares(start, stop):
    for value in range(start, stop):
        yield value ** 2

print(type(squares(0, 4))  # Prints: Generator
print(list(squares(0, 4))) # Prints: [1, 4, 9, 16]
```

Exercise

- Exercise 1 a, b
- Time: 15 minutes

Conclusions from exercise

- Generators can help to reduce the memory footprint of sequences that are consumed *directly*
- The `yield` keyword greatly simplifies the implementation of both iterators and generators.



CHALMERS
UNIVERSITY OF TECHNOLOGY

Comprehensions

- Comprehension are a special language construct that simplifies generating or transforming of sequences of elements

List comprehension:

```
# List comprehension
squares = [value ** 2 for value in range(1, 5)]
print(type(squares))            # Prints: list
```

Generator expression:

- By using parantheses `(...)` instead of brackets `[...]` Python creates a generator object instead of a list directly:

```
# Generator expression
squares_generator = (value ** 2 for value in range(1, 5))
print(type(squares_generator)) # Prints: Generator
```

CHALMERS UNIVERSITY OF TECHNOLOGY

Set and dictionary comprehension

```python
# Set comprehension
squares_set = {value ** 2 for value in range(1, 5)}
print(type(squares_set))       # Prints: set

# Dict comprehension
squares_dict = {value: value ** 2 for value in range(1, 5)}
print(type(squares_dict))      # Prints: dict
```

Exercise

- Exercise 2
- Time: 10 minutes

Conclusions from exercise

- Comprehensions are faster than explicit for loops
- Lists are highly optimized and faster than custom iterators
- Having a custom iterator class is much slower than using the `yield` keyword.

Some words of caution

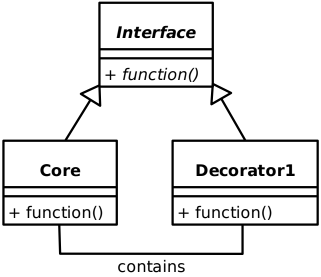
- Don’t try to optimize a certain part of your code before you don’t know that it is critical.[1]

[1] "Premature optimization is the root of all evil [...]." — Donald Knuth

The problem

- Want to dynamically modify the behavior of objects

The design pattern

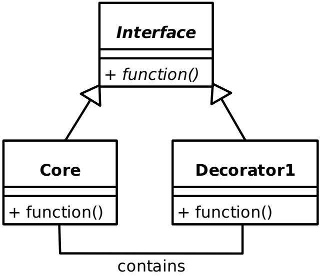


CHALMERS
UNIVERSITY OF TECHNOLOGY

Principle

- Define *wrapper* class (`Decorator1`) that delegates the core functionality to `Core` class but extends its functionality as desired.

The design pattern



A simple example

```
from abc import ABC, abstractmethod

class Greeter(ABC):
    @abstractmethod
    def greeting(self):
        pass

class English(Greeter):
    def greeting(self):
        return "hi"

class Swedish(Greeter):
    def greeting(self):
        return "hej"

class Scream(Greeter):
    def __init__(self, greeter):
        self.greeter = greeter

    def greeting(self):
        return self.greeter.greeting().upper() + "!!!"

print(Scream(English()).greeting()) # Prints: HI!!!
print(Scream(Swedish()).greeting()) # Prints: HEJ?!!!
```

CHALMERS
UNIVERSITY OF TECHNOLOGY

A simple example

- Note how the functionality can be extended and combined by simply adding a new decorator class:

```
class Question(Greeter):
    def __init__(self, greeter):
        self.greeter = greeter

    def greeting(self):
        return self.greeter.greeting() + "???"

print(Question(Scream(Swedish())).greeting()) # Prints: HEJ!!!???
```

Taking it further

- Python has *first-class functions*, i.e. functions are themselves objects
- Thus, we can generalize the Decorator pattern to function objects:

```
class LogDecorator:
    def __init__(self, function):
        self.function = function

    def __call__(self, *args, **kwargs):
        print(f"Calling {self.function.__name__}.")
        return self.function(*args, **kwargs)

logged_print = LogDecorator(print)
logged_print("hi") # Prints: Calling print. hi.
```

… and further

- The code for this can be simplified by using the closures to store the data required by the `logger` function:

```
def log_decorator(function):
    def logger(*args, **kwargs):
        print(f"Calling {function.__name__}.")
        function(*args, **kwargs)
    return logger

logged_print = log_decorator(print)
logged_print("hi") # Prints: Calling print. hi.
```

Wait a minute . . .

- **Closure of a function**: The variables defined in the scopes surrounding the function definition.

```
def say_something_factory(something):
    def say_something():
        print(something)
    return say_something

say_something = say_something_factory("Hi from the closure.")
say_something() # Prints: "Hi from the closure."
```

When a function is defined within another function, the local variables of the enclosing function are stored in the closure of the nested function.

CHALMERS UNIVERSITY OF TECHNOLOGY

Decorators in Python

- Python provides the `@decorator` specieal syntax to apply decorators to functions (and classes)

```
@log_decorator
def my_print(what):
    print(what)

my_print("hi") # Prints: Calling print. hi.
```

- This is just *syntactic sugar* for:

```
my_print = log_decorator(my_print)
```

Exercise

- Exercise 3
- Time: 10 Minutes

Decorators in Python

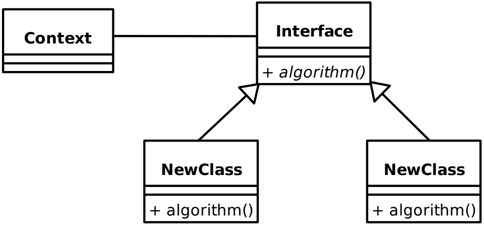
- This example only illustrated the basic use of decorators in Python
- We will come back to them later.

CHALMERS
UNIVERSITY OF TECHNOLOGY

The problem

- Let user of a class choose the specific algorithm used in a computation

The design pattern



Example

- Reducing a list of numbers

```python
from abc import ABC, abstractmethod

class Reduction:
    def __init__(self, list, strategy):
       self.list = list
       self.strategy = strategy

    def compute(self):
        return self.strategy(list)

# Could simply use collections.abc.Callable
class ReductorInterface(ABC):
    @abstractmethod
    def __call__(self, list):
        pass
```

CHALMERS
UNIVERSITY OF TECHNOLOGY

Example

```
class Sum(ReductorInterface):
    def __call__(self, list):
       return sum(list)

class Product(ABC):
    def __call__(self, list):
        result = 1
        for item in list:
            result *= item
        return result

list = [1, 2, 3, 4]
sum_reduce = Reduction(list, Sum())
print(sum_reduce.compute())      # Prints: 10

product_reduce = Reduction(list, Product())
print(product_reduce.compute()) # Prints: 24
```

Simplifications

- Since Python has first-class functions, we don't need to write classes:

```
def product(list):
      result = 1
      for item in list:
          result *= item
      return result

sum_reduce = Reduction(list, sum)
print(sum_reduce.compute())      # Prints: 10

product_reduce = Reduction(list, product)
print(product_reduce.compute()) # Prints: 24
```

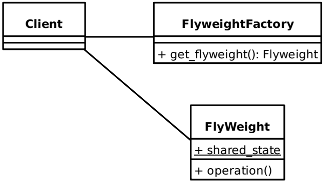
Why not just an `if` statement?

- *Open-closed principle*: software entities should be open for extension, but closed for modification
- The Strategy pattern allows changing the behavior of the `Context` class without changing any of its code.
- Documentation: The abstract interface for the strategy classes acts as language-level documentation of how to extend the code

The problem

- Frequent access to a memory-heavy object causes memory issues

The design pattern



Factory classes

```
black = Color("#000000")

class ColorFactory:
    @staticmethod
    def black():
        return black
```

- A factory class is a specialized class tasked with creating objects of another class
- The constructor methods in the factory class are typically static methods.
- Why an extra class?
  - Single-responsibility principle: Each class should have a single responsibility
  - When programmers see a factory class, they immediately know its role

Python implementation

- Can use `__new__` method instead of explicit factory class.

```
class Color:
    _colors = {}
    def __new__(cls, color_string):
        if color_string in Color._colors: # Check if specific color already exists.
            return _colors[color_string]

        new_color = super().__new__(cls)  # Create new object by calling __new__ of super()
        _colors[color_string] = new_color # Store newly created object for next call.
        self.initialized = False          # Flag that object has not yet been initialized.
        return new_color

    def __init__(self, color_string):
        if not self.initialized:          # Guard against multiple initialization.
            self.color_string = color_string
            self.initialized = True
```

```python
def __new__(cls, *args, **kwargs):
```

Understanding `__new__`:

- Called *before* `__init__` method
- Static method by default (no need for `@staticmethod` decorator)
- First argument `cls`: The class of the object being constructed
- Remaining arguments: Other arguments passed to constructor.
- Should return the newly constructed object.

Comments

- Note that we had to guard against multiple initialization in `__init__` function.
- Use this only if you know that memory is an issue. Helps only if objects are in fact identical, i.e. if *a lot of* `black` objects would be created.[2]

```
black = Color("#000000")
other_black = Color("#000000")
red = Color("#FF0000")

Print(black is other_black) # Prints: True
Print(red is other_black)   # Prints: False
```

[2] Another way to reduce the memory footprint of Python object is to use `__slots__`.

Exercise

CHALMERS
UNIVERSITY OF TECHNOLOGY

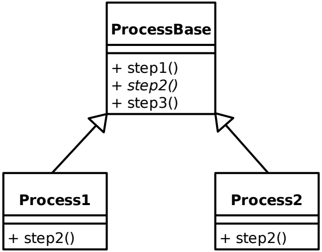
- Exercise 4
- Time: 10 minutes

Agenda

The problem

- Two processes share common steps

The pattern



CHALMERS
UNIVERSITY OF TECHNOLOGY

Example: File processing

```
class FileProcessorBase:
    def __init__(self,
                input_file,
                output_file):
        self.input_file = input_file
        self.output_file = output_file

    def read(self):
        self.content = open(self.input_file).read()

    @abstractmethod
    def process(self):
        pass

    def write(self):
        with open(self.output_file, "w") as file:
            file.write(self.content)

    def execute(self):
        self.read()
        self.process()
        self.write()
```

Example: File processing

```python
class UpperCaseTransformer(FileProcessorBase):
    def __init__(self,
                 input_file,
                 output_file):
        super().__init__(input_file, output_file)

    def process(self):
        self.content = self.content.upper()

open("test.txt", "w").write("This is just a test.")
transformer = UpperCaseTransformer("test.txt", "TEST.txt")
transformer.execute()
```

Advantages

- DRY principle: Code for common steps can be reused
- Open/Closed principle: Functionality can be easily extended without the need to modify any of the existing classes

CHALMERS
UNIVERSITY OF TECHNOLOGY

Iterators and generator

- Iterator pattern
- Language support for iterators and generators (`for`, `yield`, . . .)

Decorators

- The decorator pattern
- Language support for decorators

Strategy pattern

- Supports open-closed principle leading to modular code.

Flyweight pattern

- Using static variables to save memory
- Manipulating object construction in Python

More design patterns:

- There are a lot more.[3]

[3] See `https://en.wikipedia.org/wiki/Software_design_pattern`

Why use design patterns:

- Duck typing in Python allows most patterns to be implemented in a less formal way.
- **But**:
  - Being explicit about the design can make code easier to understand
  - Design patterns are known across programming languages