bianca.phelan@tudublin.ie

O Object
O Oriented
P Programming

S1 = Python with Bianca

S2 = Java with Bryan

DT228(TU856)/DT282(TU858) - 2

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
TU DUBLIN
TECHNOLOGICAL UNIVERSITY DUBLIN

COMPUTER SCIENCE

This is the last lecture before the Christmas break.

You can always contact me throughout the year if you have a Python question or a question on the Python component of the exam.

All course content will be available in the gitHub repository from next week on.

Objectives

- Discover design patterns
- Discuss some of the most common design patterns and analyse different implementation strategies

What is a Design Pattern?

- **Proven** solution to **common** problems
  - Specific context
- It's a description, not ready made source code (although examples are available)
- Illustrates effective programming in specific scenarios
  - They can also explain why other solutions might not work
  - Speed up the development process
  - Facilitate well-designed code
  - Anticipate any future issues that there might be
  - **Can considerably improve your life as a programmer**

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Where does it come from?

- In the 90s 4 programmers got together and formulated the most common patterns of problems and formulated solutions for them
- “Design Patterns: Elements of Reusable Object-Oriented Software” aka “The Gang of Four (GoF)”

Different Types of Design Patterns

- Behavioural Patterns
  - Example: Iterator, State, Observer
- Structural Patterns
  - Example: Facade, Decorator, Adapter
- Creational Patterns
  - Example: Singleton

Example Patterns in Python:

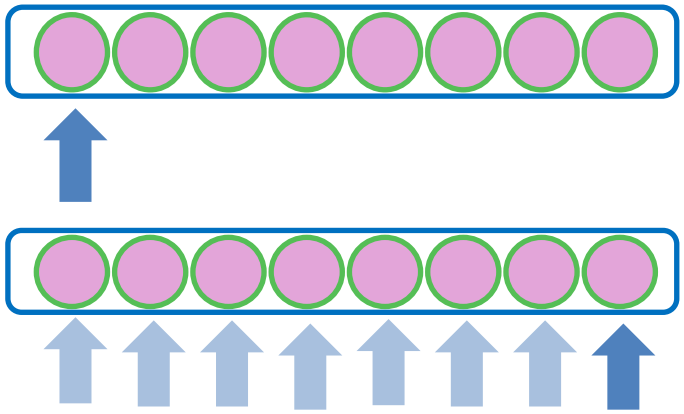
- Model View Controller Pattern
- Singleton pattern
- Factory pattern
- Builder Pattern
- Prototype Pattern
- Facade Pattern
- Command Pattern
- Adapter Pattern
- Prototype Pattern

And more…...

Iterator

Behavioural Design Pattern

Behavioural Pattern - Iterator



- **Iterator** pattern allows to traverse a collection without exposing any of the internal workings
- Use Case: Standard way of traversing collections
- In a programming language without patterns, an iterator would have a `next()` method and a `done()` method, and the iterator loops across all the containers using these methods.

```
WHILE NOT(iterator.done())
    DO item = iterator.next()
    # do more stuff
ENDWHILE;
```

Iterator in Python

[1]

Name	Description
_ _next_ _()	The next method returns the next element from the container.
StopIteration()	The StopIteration is an exception that is raised when the last element is reached.
_ _iter_ _()	The iter method makes the object iterable, and returns an iterator.

Iterator in Python

[4]

- Python supports the iterator pattern in the most basic form: it's built into the language
- Python's for loop abstracts the iterator pattern so thoroughly, that most people are not even aware of using a pattern!

```
my_numbers = [1,2,3,4,5]

for number in my_numbers:
    print(number)
```

```
DesignPatterns
/Users/bi
1
2
3
4
5
```

Under the hood there's a design pattern at work!

Explicitly using the Iterator in a Python Example from Previous Slide

- Re-enacting the example from the previous slide using a pattern explicitly

```
it = iter(my_numbers)

print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
DesignPatterns
1
2
3
4
5
```

- And with one too many:

```
DesignPatterns
/Users/bianca.schoenphelan/PycharmPro
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Py
    print(next(it))
StopIteration
```

Cont'd

- Of course, for loop doesn't explicitly write out every step, but uses a while true underneath

```
it = iter(my_numbers)
while True:
    try:
        number = next(it)
    except StopIteration:
        break
    else:
        print(number)
```

Objects that are their own Iterator

- every time you `iter()` over a list, dictionary or set you receive a new iterator object that visits the container item from the start over again each time
- Not all Python objects behave this way
- Files are different!
  - Traverses lines in a file
  - A new for loop doesn't start at the beginning but where a previous iteration had left off

Revision on Files

```
def parse_email(f):
    envelope = ''
    print(id(envelope))
    for line in f:
        envelope.join(line)
        break
    headers = {}
    for line in f:
        if line == '\n':
            break
        name, value = line.split(':', 1)
        headers[name.strip()] = value.lstrip().rstrip('\n')
    body = []
    for line in f:
        if line.startswith('From'):
            break
        body.append(line)
    return envelope, headers, body
```

```
with open('email.txt') as f:
    envelope, headers, body = parse_email(f)

print(headers['To'])
print(len(body), 'lines')
```

- Three for loops parse this file
  - Envelope is a single line
  - Header is a series of colon separated names and values that are terminated by a single blank line
  - Body last, ends with either next envelope or the end of the file

The File Object Preserves the State

- The file object preserves state
- Look at `iter()` instead of the for loop and see

```
f = open('email.txt')
print (f)
it1 = iter(f)
print(it1)
it2 = iter(f)
print(it2)

if(it1 is it2 is f):
    print("yes")
else:
    print("no")
```

```
DesignPatterns
/Users/bianca.schoenphelan/PycharmProjects/TestProject1/venv/bin
<_io.TextIOWrapper name='email.txt' mode='r' encoding='UTF-8'>
<_io.TextIOWrapper name='email.txt' mode='r' encoding='UTF-8'>
<_io.TextIOWrapper name='email.txt' mode='r' encoding='UTF-8'>
yes
```

- `iter()` has to return an iterator
- This iterator can be an iterable itself!!!
- Instead of creating a new iterator object, the file itself acts as an iterator!
- First consumer to want `next()` will be served

Files and iter()

[4]

- Rule says that iter() must return an iterator, never says that the iterator cannot be the object itself!
- Files return themselves as an iterator
- So instead of creating a separate Iterator object, and yields to a single continuous series of lines
- You could write this behaviour yourself, see [4]

Creating Iterators in Python

[2]

To create the iterator object

To traverse the container

- 2 Parts: (1) An iterable class, and (2) An iterator class
- A class can implement Iterator and plug in to Python's native iteration mechanisms using `for`, `next()`, `iter()`
- Must offer `__iter__()` method that returns the iterator
  - The container must be an iterable
- Must offer `__next__()` method that offers the next object and throws `StopIterator` exception
- Also needs an `__iter__()` method that returns itself for those who want to use the container in a for loop

Implementation Example: Your own Iterator

```
class OddIterator(object):
    "An iterator."

    def __init__(self, container):
        self.container = container
        self.n = -1

    def __next__(self):
        self.n += 2
        if self.n > self.container.maximum:
            raise StopIteration
        return self.n

    def __iter__(self):
        return self
```

```
class OddNumbers(object):
    "An iterable object."

    def __init__(self, maximum):
        self.maximum = maximum

    def __iter__(self):
        return OddIterator(self)
```

```
numbers = OddNumbers(7)

for n in numbers:
    print(n)
```

```
DesignPatte
/Users/
1
3
5
7
```

```
it = iter(OddNumbers(5))
print(next(it))
print(next(it))
```

```
DesignPatt
/Users
1
3
```

```
print(list(numbers))
print(set(n for n in numbers if n > 4))
```

```
DesignPatterns
/Users/bianca
[1, 3, 5, 7]
{5, 7}
```

Singleton

Creational Design Pattern

Singleton

- Restricts a class to just **one** instance
- Provides a global access point to this one instance
- Helps to manage shared resources, such as databases, files, printer that would all be shared by multiple applications
- Stores a global state
- Creates a simple logger
- Can be difficult to test in unit testing, as most tests use an inheritance principle when creating mock objects for testing.
- Interferes with parallel programming, distributed computing

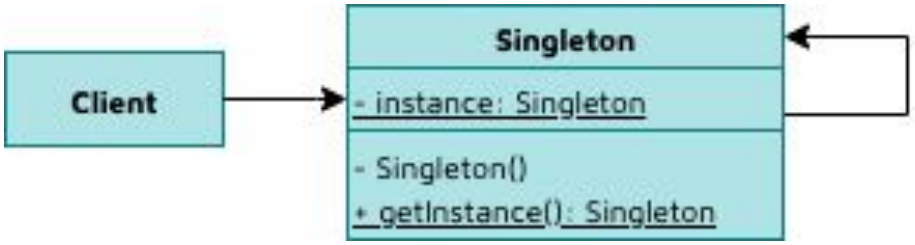
Singleton

- In Python sometimes considered an “anti-pattern”
- Often used by people who come from more restrictive languages
  - Often considered that you’re doing something wrong if you’re trying to solve a problem this way?
- Why discuss it so?
  - One of the most famous design patterns
  - The idea is useful, even if not very Pythonic
  - Python provides in-built Singletons: True, False, None
- In most programming languages implemented by making the constructor private
  - Python doesn’t do “private” like other languages do, we use `__new__`

What does `__new__` do?

- It creates a new instance of that class
- Sounds a lot like `__init__`
  - `__new__` handles object creation
  - `__init__` handles object initialisation
  - Both are defined in object and you override them if you want custom behaviour
  - `__new__` is called first for creation, then `__init__` for initialisation of a created object
- We override it
  - First check if it has been created
  - If not, we create it

Structure of a Singleton



[2]

```
class Singleton:
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance


s = Singleton()
print("Object created:", s)
s1 = Singleton()
print("Object created:", s1)
```

```
/Users/bianca.schoenphelan/Documents/OOP_Class/Code/venv/bin/pyt
Object created: <__main__.Singleton object at 0x7fb74b1070d0>
Object created: <__main__.Singleton object at 0x7fb74b1070d0>

Process finished with exit code 0
```

[9]

Also often Implemented with a decorator!

1

```
class Singleton:
    def __init__(self, cls):
        self._cls = cls

    def Instance(self):
        try:
            return self._instance
        except AttributeError:
            self._instance = self._cls()
            return self._instance

    def call(self):
        raise TypeError

    def __instancecheck__(self, instance):
        return isinstance(instance, self._cls)
```

2

```
@Singleton
class DBConnection:
    def __init__(self):
        print("established db connection")

    def __str__(self):
        return "Database connection object"
```

Singleton via decorator cont'd

```
db_connection1 = DBConnection.Instance()
db_connection2 = DBConnection.Instance()

print(f"ID of connection 1: {id(db_connection1)}")
print(f"ID of connection 2: {id(db_connection1)}")
```

```
/Users/bianca.schoenphelan/Documents/
established db connection
ID of connection 1: 140722315490304
ID of connection 2: 140722315490304
```

Singleton via decorator cont'd

```
db_con = DBConnection()    #causes the TypeError
```

```
/Users/bianca.schoenphelan/Documents/OOP_Class/Code/venv/bin/python /User
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/Code/tutorial.py",
    db_con = DBConnection()
TypeError: 'Singleton' object is not callable

Process finished with exit code 1
```

Revision + Write your own Decorators

Functions in Python

```
def say_name():
    print("Bianca")


def say_nationality():
    print("German")


def say(function):
    return function


say(say_name)()
say(say_nationality)()
```

```
/Users/bianc
Bianca
German
```

- In Python functions are first-class objects
  - Functions are objects (they can be referenced to, can be passed as a variable, and can be returned from other functions)
  - Functions can be defined inside another function

Inner Function in Python

```
def say():

    def say_name():
        print("Bianca")

    def say_nationality():
        print("German")

    say_name()
    say_nationality()

say()
```

The output will be the same as before.

Now with decorator

```
def say(func):
    def say_name():
        print("Bianca")

    def say_nationality():
        print("German")

    def wrapper():
        say_name()
        say_nationality()
        func()

    return wrapper

@say
def start_example():
    print("In the example")
```

```
/Users/bianca.s
Bianca
German
In the example
```

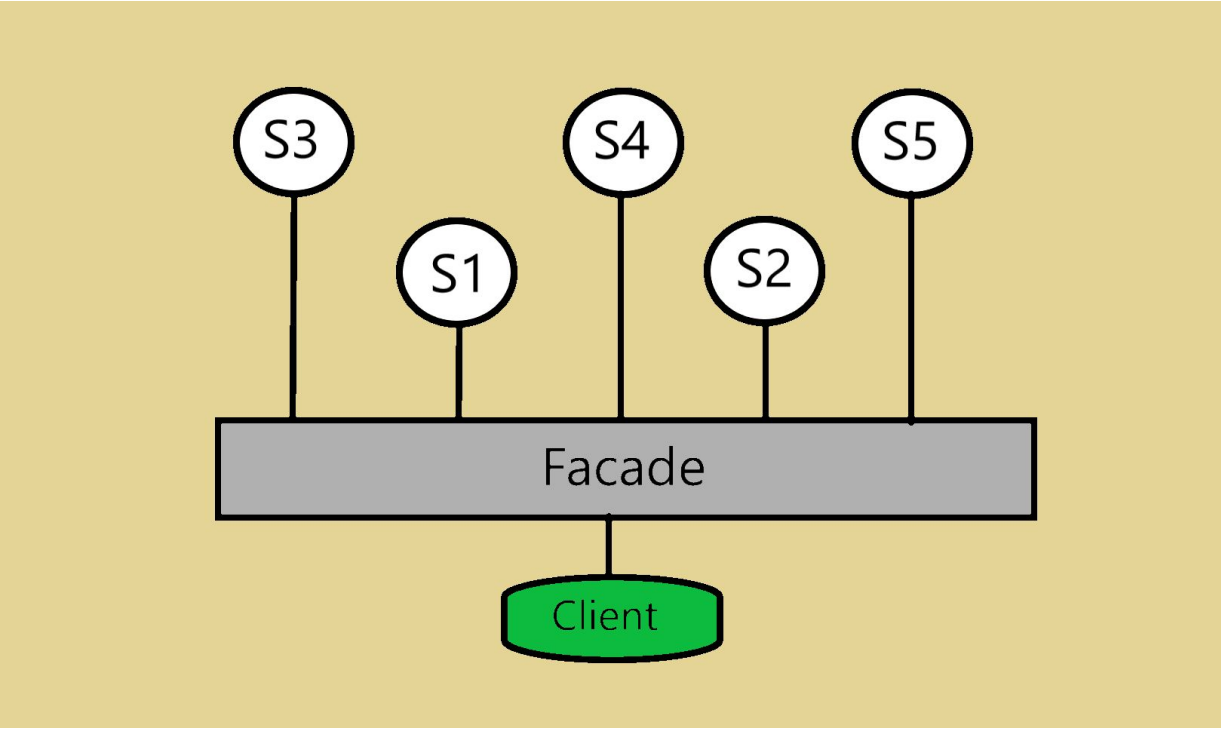
- When start_example() is called it goes straight to the say() function and defines say_name() and say_nationality() and the wrapper() function and finally returns the wrapper

Facade

Structural Design Pattern

What does Facade do?

- Hides system complexity
- Provides one simple interface



[10]

Understanding Facade

[10]

- Metaphor of a shopkeeper who knows where everything is, but customers do not, so they ask the shopkeeper for the products
  - The Shopkeeper acts as the Facade of the shop
- Facade provides a simple and easy to understand interface to complex systems
  - Facade Class: Implements the interface that is being used by clients
  - System Classes: Multiple system classes with complex behaviours and interactions
  - Client Classes: Multiple client classes wanting to use the system and via the easy to use facade interface class

Conditions on Using Facade

[10]

- The multiple sub-systems should not depend on each other
- System should have sufficient complexity to make accessing and using very hard (our example will be simple)
- Try to provide one interface for all functionalities

Benefits of Facade [10]

- Even if there are multiple changes done to the sub-system over its lifespan, the interface to the clients remains the same
- The facade collects all sub-system classes and provides a meaningful interface
- Sub-systems are not aware of the Facade
  - Another design pattern “Mediator” is like Facade, but the sub-system is aware of it

Python Facade Example: Cooking

```python
class Cutter:

    def cut_vegetables(self):
        print("Cutting veggies!")


class Boiler:

    def boil_vegetables(self):
        print("Boiling veggies!")


class Frier:

    def fry_vegetables(self):
        print("Frying veggies!")
```

The complex system!

The Facade Easy Interface!

```python
class Cook:

    def prepare_dish(self):
        self.my_cutter = Cutter()
        self.my_cutter.cut_vegetables()


        self.my_boiler = Boiler()
        self.my_boiler.boil_vegetables()


        self.my_frier = Frier()
        self.my_frier.fry_vegetables()
```

Example cont'd

[10]

```
class Cutter:

    def cut_vegetables(self):
        print("Cutting veggies!")


class Boiler:

    def boil_vegetables(self):
        print("Boiling veggies!")


class Frier:

    def fry_vegetables(self):
        print("Frying veggies!")
```

```
class Cook:

    def prepare_dish(self):
        self.my_cutter = Cutter()
        self.my_cutter.cut_vegetables()


        self.my_boiler = Boiler()
        self.my_boiler.boil_vegetables()


        self.my_frier = Frier()
        self.my_frier.fry_vegetables()
```

```
my_cook = Cook()
my_cook.prepare_dish()
```

Usage and output.

```
/Users/bianca.schoe
Cutting veggies!
Boiling veggies!
Frying veggies.
```

Summary

- ★ **Design Patterns**
- ★ **Iterator Design Pattern**
- ★ **Singleton Design Pattern**
- ★ **Facade Pattern**

References

1. Damian Gordon, Iterator Pattern. DIT.
2. Implementation of Top Design Patterns in Python, Lisa Plitnichenko, 8 Nov 2020, via Medium https://medium.com/python-pandemonium/top-design-patterns-in-python-9778843d5451, accessed Dec 2020.
3. Python 3: Object Oriented Programming, Dusty Phillips, 2nd edition, 2015
4. Python Pattern Guide, Brandon Rhodes, 2018. https://python-patterns.guide/ accessed Dec 2020.
5. Decorators in Python, Goutom Roy, 26 May 2019, Medium, https://medium.com/better-programming/decorators-in-python-72a1d578eac4, accessed Dec 2020.
6. Decorator Pattern, Brandon Rhodes, 2018-2020, https://python-patterns.guide/gang-of-four/decorator-pattern/, accessed Dec 2020
7. Tutorialspoint, Decorator Pattern, https://www.tutorialspoint.com/python_design_patterns/python_design_patterns_decorator.htm, accessed Dec 2020
8. Learn object-oriented programming in Python, 15 August 2015, Eduonix, https://blog.eduonix.com/software-development/learn-object-oriented-programming-in-python-composition/, accessed Dec 2018
9. Singletons in Python, Goutom Roy, 26 May 2019, Medium, https://medium.com/better-programming/singleton-in-python-5eaa66618e3d, accessed Dec 2020.
10. Facade Design Pattern, Hardik Patel, 29 March 2019, https://aaravtech.medium.com/design-patterns-in-python-facade-65b8a393ff68, accessed Dec 2020.