Digital Career Institute

Python Course - OOP in Practice

Digital Career Institute

Goal of the Module

The goal of this submodule is to help the student understand advanced principles and design patterns of Object Oriented Programming (OOP). By the end of this submodule, the learners will be able to understand:

- The singleton pattern.
- The factory pattern.
- Magic methods.

Topics

- Singleton pattern
- Factory design method
- Dunder and magic methods

Digital Career Institute

OOP Design Patterns

Design Patterns

When designing an application, it’s worth checking if a solution to a similar problem already exists instead of always inventing a new one: **Don’t reinvent the wheel.**

In object oriented programming (OOP), there are design patterns that help us with the conception of our classes.

What are Design Patterns?

A design pattern is a way of implementing a solution to a problem.

In OOP, design patterns are ways of defining and using classes.

Design patterns are often not provided as implemented types of a language and it is the programmer who has to ensure the class is designed following the pattern properly.

Design Patterns in OOP

Digital Career Institute

In OOP, there are three categories of design patterns:

Creational DPs:

Describe how to create objects.

- Factories
- Singletons
- Builder
- ...

Structural DPs:

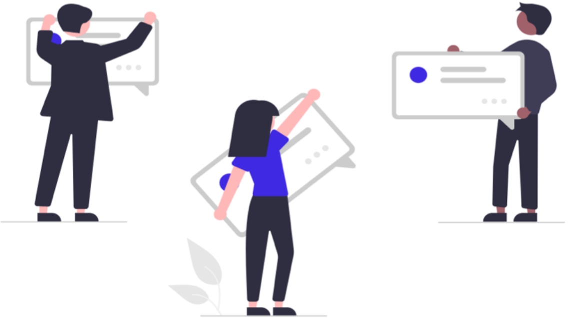
Show how to tie objects together to form larger structure.

- Adapter
- Decorators
- Composite
- ...

Behavioural DPs:

Describe how related objects communicate with each other and what their behavior is towards others.

- Iterator
- Command
- Observer
- ...



Expert Round

Digital Career Institute

[UNDRAW]

Topics:

- Single responsibility principle
- Factories

Singletons

What is a Singleton?

- A singleton pattern is a class where only one instance is ever created.
- Because of this, they are often used to provide the same object to any part of the code.
- This means that they act as a global scope manager.
- There are different ways to implement a singleton.

Pros & Cons

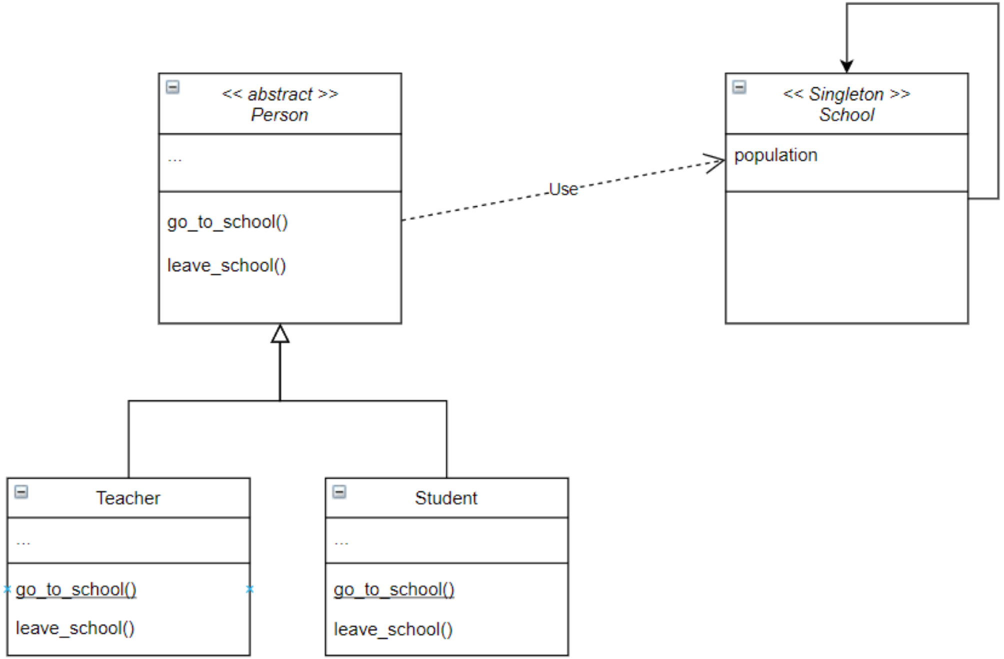
Pros	Cons
+ It saves time if the instantiation of the object takes a lot of time.	- It makes testing more difficult.
+ It saves memory, since only one instance is created. Especially in a case when the code will instantiate a class multiple times.	- The code can be difficult to maintain and errors might be harder to debug (Who changed the state of the singleton object? when and where?).

Example

- Consider a scenario where there is only one school and many students and teachers in that school.

- The number of people in the school **‘population’** depends on who went to school and who left.

- In this scenario it makes sense to model the **School** as a singleton since there is only one instance of it.



Implementing Singletons in Python

school/classes.py

```
class School:
    class __School:
        def __init__(self, population=None):
            self.population = population

        def __str__(self):
            return str(self.population)

    __instance = None

    def __new__(cls, population):
        if not cls.__instance:
            cls.__instance =
cls.__School(population)
        return cls.__instance
```

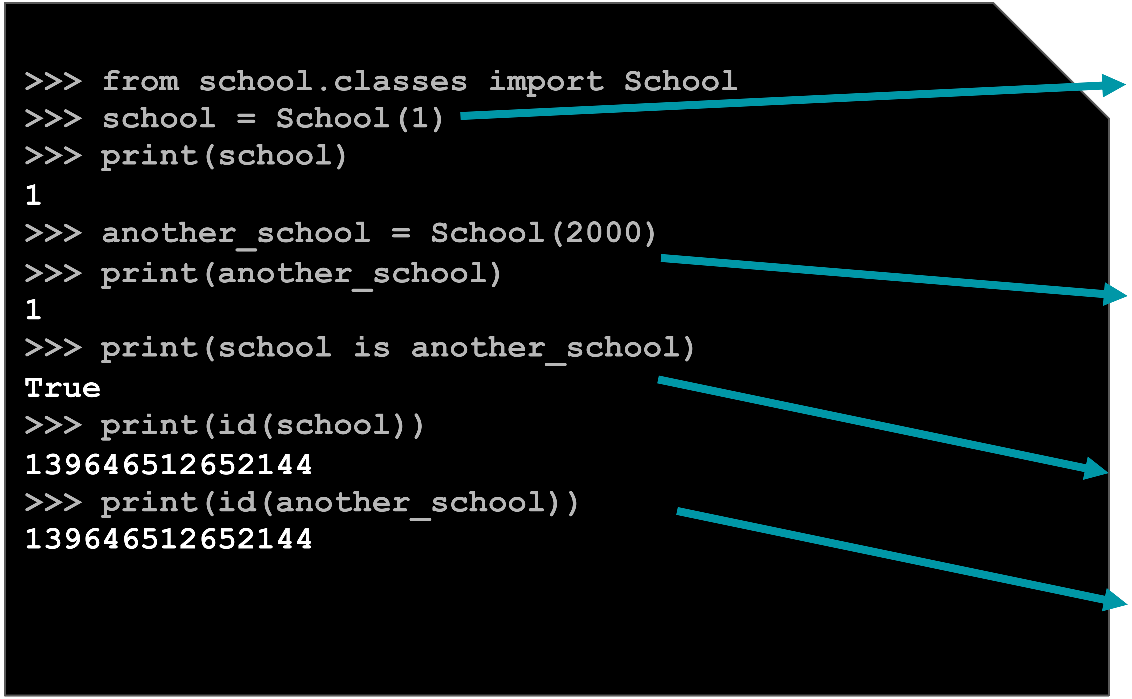
The **__Singleton** class contains the logic of our target class.

The property **__instance** will hold the object instantiated. It is set to **None** on initialization.

The class method **__new__** runs before the object method **__init__** whenever an object is instantiated.

In this implementation, the constructor takes an argument that is only used the first time.

Implementing Singletons in Python



```
>>> from school.classes import School
>>> school = School(1)
>>> print(school)
1
>>> another_school = School(2000)
>>> print(another_school)
1
>>> print(school is another_school)
True
>>> print(id(school))
139646512652144
>>> print(id(another_school))
139646512652144
```

In the previous implementation, creating a singleton object for the first time, will create a new object with the input argument.

If the singleton gets instantiated again, the new argument takes no effect.

This is because the constructor returned exactly the same object.

The internal id of both objects is the same.

OOP Design Patterns: Factories

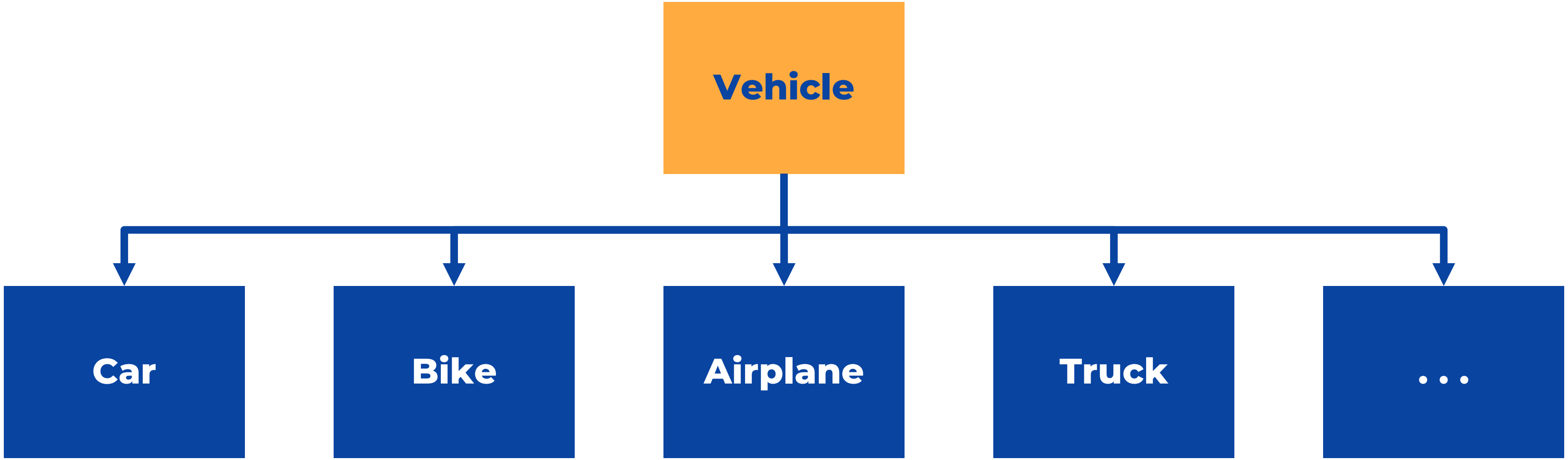
Digital Career Institute

Factory Method - Introduction

- A way of designing class so that object attributes and methods are defined only at runtime.

- Why?
  - Sometimes, the class-based design requires objects to be created in response to conditions that can’t be predicted when a program is written *(Lutz, Learning Python).*

Class Factories



Sometimes there are many classes that may need to be created, all with similar properties and different values.

Sometimes, a new class needs to be created on runtime with input parameters.

Class Factories

The most basic implementation of a class factory is a function that creates a new class on runtime.

The property values can be passed on to the factory.

vehicle/classes.py

```python
def vehicle_factory(has_wheels, num_wheels):
    class Vehicle:
        def __init__(self, **kwargs):
            self.has_wheels = has_wheels
            self.num_wheels = num_wheels
            self.properties = kwargs
    return Vehicle
```

```
>>> from vehicle.classes import vechicle_factory
>>> Car = vehicle_factory(True, 4)
>>> my_car = Car(brand="Skoda")
>>> print(my_car.num_wheels)
4
>>> print(my_car)
<__main__.vehicle_factory.<locals>.Vehicle object
at 0x7f04e608ca60>
```

This design pattern allows the definition of any class of vehicle on runtime.

The objects created by this class are of the type `Vehicle`.

Class Factories

A class can also be created using the `type` constructor inside the factory function.

This function creates a class with the given name, extending the given object and having the given properties and methods.

vehicle/classes.py

```python
def vehicle_factory(name, has_wheels,
                    num_wheels):
    def init(self, **kwargs):
        self.properties = kwargs
    return type(name, (object,), {
        "__init__": init,
        "has_wheels": has_wheels,
        "num_wheels": num_wheels
    })
```

```
>>> from vehicle.classes import vechicle_factory
>>> Car = vehicle_factory("Car", True, 4)
>>> my_car = Car(brand="Skoda")
>>> print(my_car)
<__main__.Car object at 0x7f396fb28fd0>
```

The objects created using `type` are of the type indicated when calling the factory.

Factory Method

Very often, a class factory is used to have a common interface to object creation.

Depending on the given input parameter, the factory will return one or another pre-defined class.

The factory method is also sometimes used with classes instead of functions.

classes.py

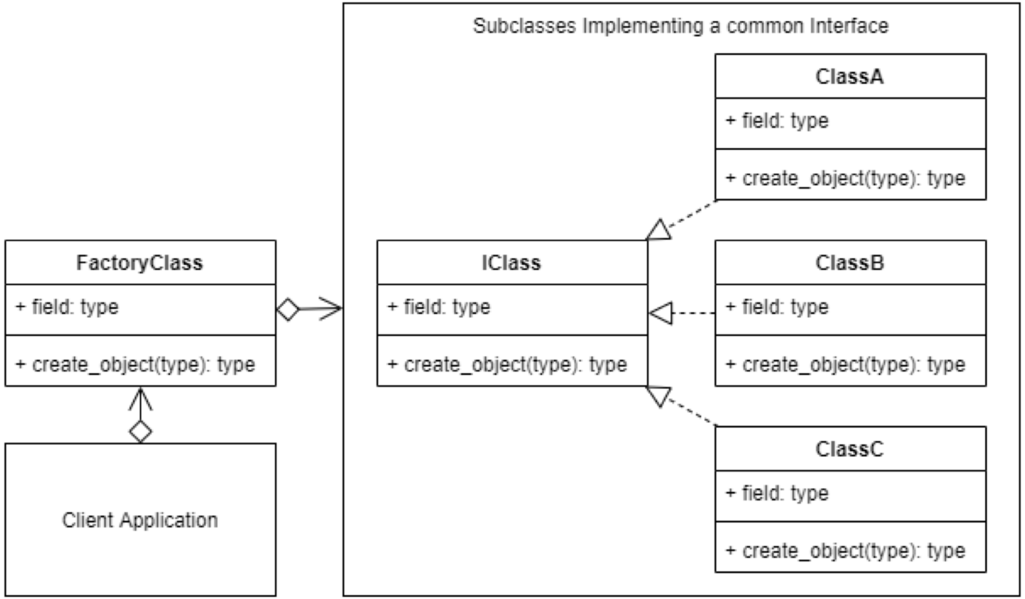
```
def factory(type):
    if type == "Car":
        return Car
    if type == "Bike":
        return Motorbike
    if type == "Airplane":
        return Airplane
    if type == "Whale":
        return Whale
```

classes.py

```
def factory(type):
    options = {
        "Car": Car,
        "Bike": Motorbike,
        "Airplane": Airplane,
        "Whale": Whale,
    }
    return options[type]
```

This factory pattern is named the **Factory Method**.

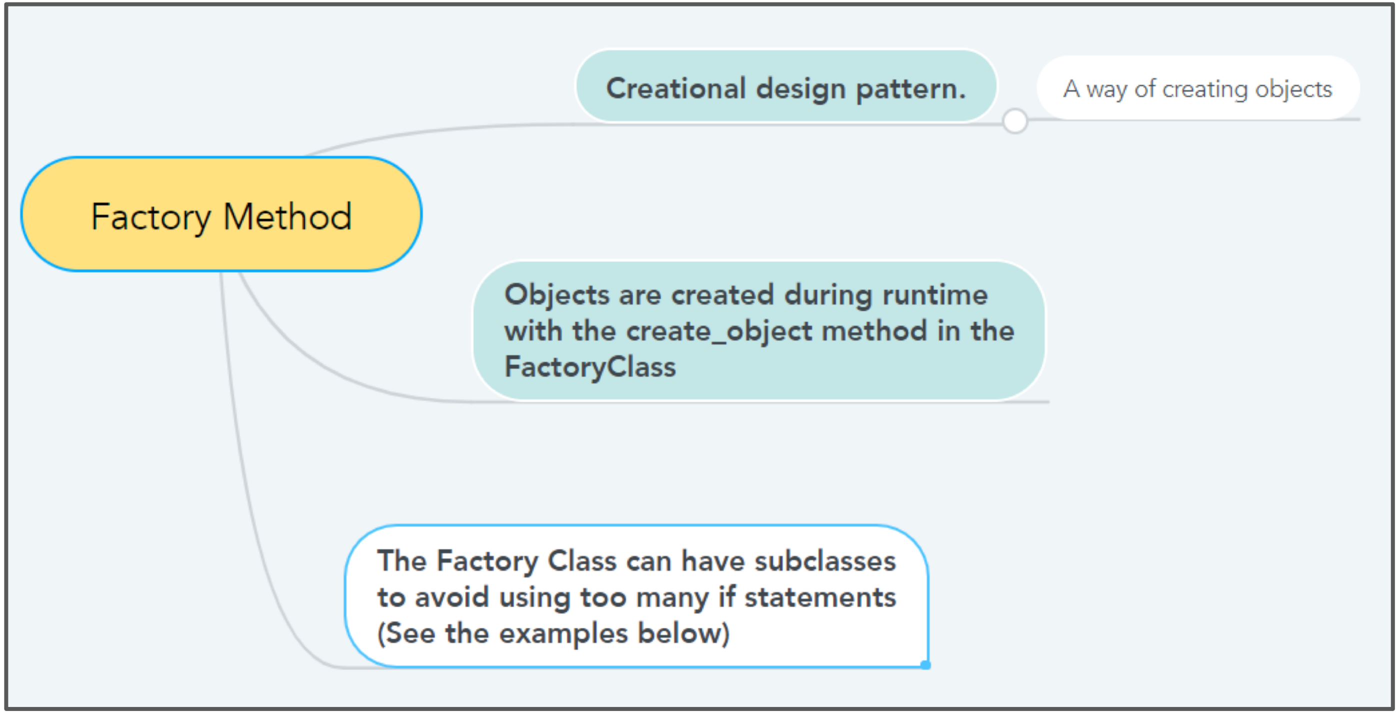
Factory Method - Explanation


Subclasses Implementing a common Interface
ClassA
+ field: type
+ create_object(type): type
FactoryClass
+ field: type
+ create_object(type): type
IClass
+ field: type
+ create_object(type): type
ClassB
+ field: type
+ create_object(type): type
ClassC
+ field: type
+ create_object(type): type
Client Application

UML Design for factory design. Source: https://medium.com/design-patterns-in-python/factory-pattern-in-python-2f7e1ca45d3e

- `IClass` **is an abstract class.**
- `ClassA`, `ClassB` **and** `ClassC` **are its subclasses.**
- **The** `FactoryClass` **contains the method** `create_object()`.
- **During runtime, and depending on the client choices, the** `FactoryClass` **creates an object of** `ClassA`, `ClassB` **or** `ClassC`.

Factory Method - Notes



Factory Method

Creational design pattern.

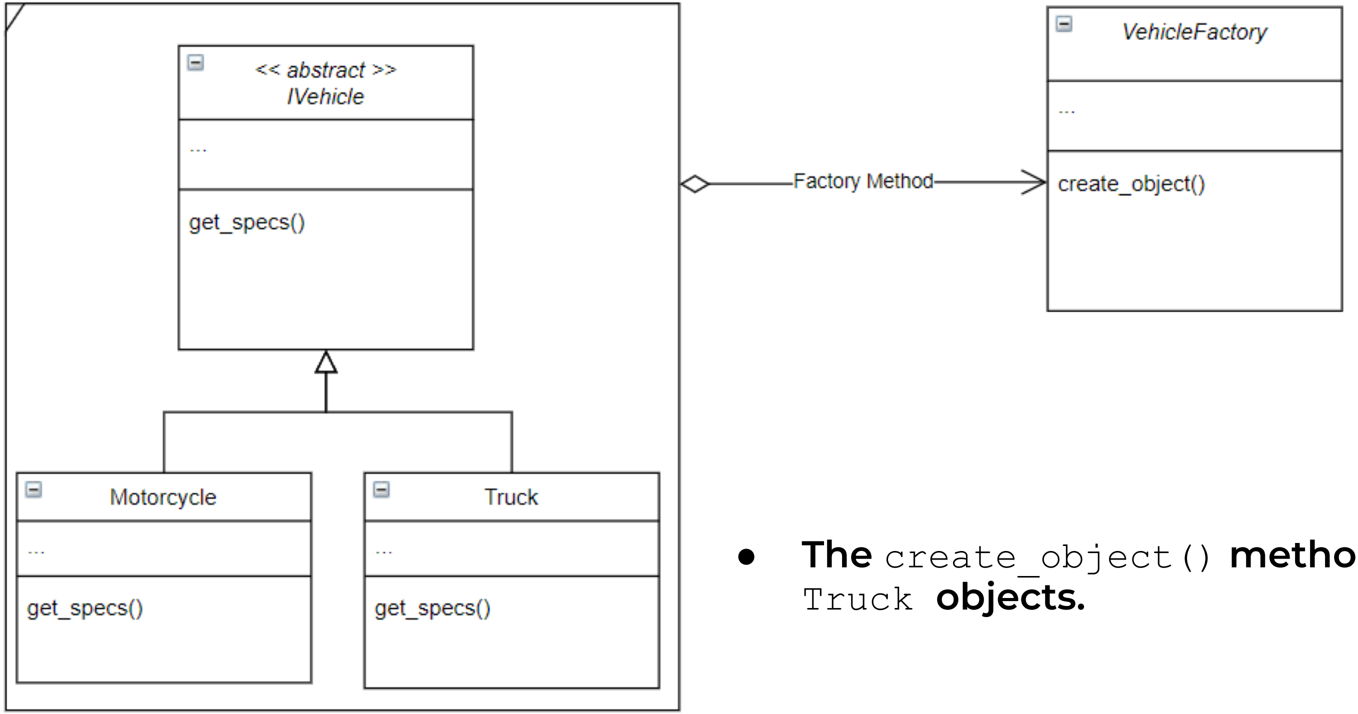
A way of creating objects

Objects are created during runtime with the create_object method in the FactoryClass

The Factory Class can have subclasses to avoid using too many if statements (See the examples below)

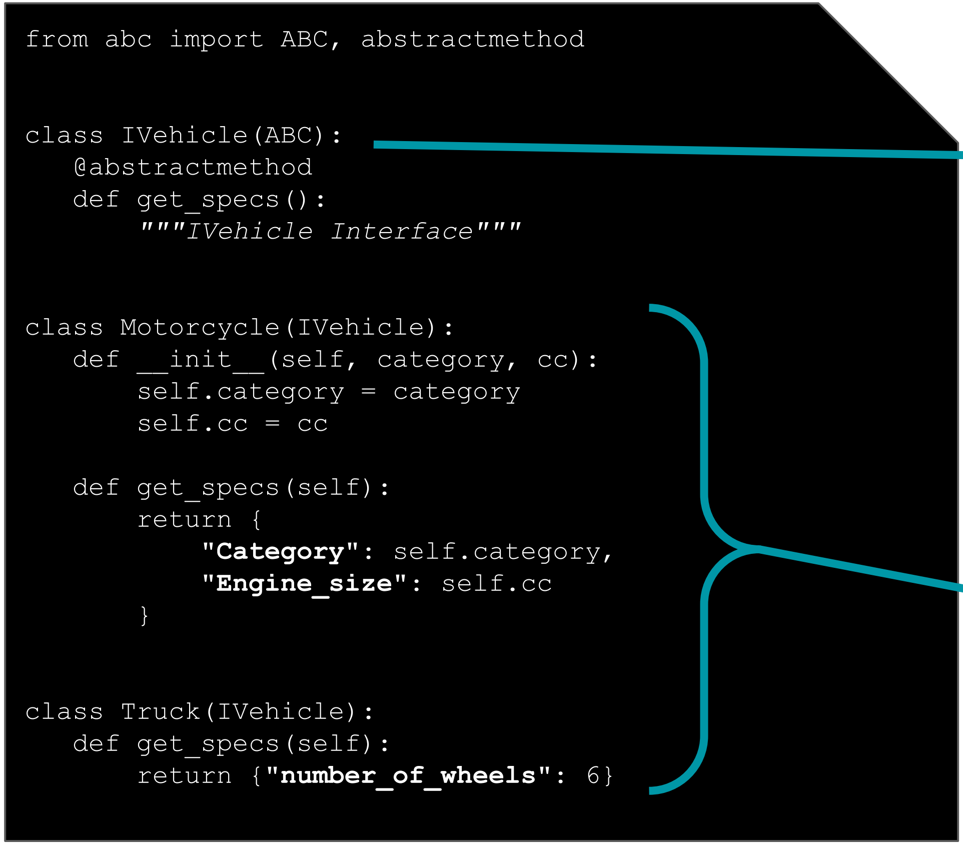
Factory Method - Example, Part 1

Consider the following class diagram:



- **The** `create_object()` **method allows** `Motorcycle` **and** `Truck` **objects.**

Factory Method - Example, Part 1



```
from abc import ABC, abstractmethod


class IVehicle(ABC):
   @abstractmethod
   def get_specs():
       """IVehicle Interface"""


class Motorcycle(IVehicle):
   def __init__(self, category, cc):
       self.category = category
       self.cc = cc

   def get_specs(self):
       return {
           "Category": self.category,
           "Engine_size": self.cc
       }


class Truck(IVehicle):
   def get_specs(self):
       return {"number_of_wheels": 6}
```

Code parts

Abstract class.

Subclasses that will be instantiated at run time.
Note: The use of the factory method is to allow the user to decide what objects to instantiate at runtime.

Factory Method - Example, part 2

```
class VehicleFactory:
   @staticmethod
   def create_object(vehicle_type, *args, **kwargs):
       try:
           if vehicle_type == "Motorcycle":
               return Motorcycle(*args, **kwargs)
           elif vehicle_type == "Truck":
               return Truck()
           else:
               raise AssertionError("Vehicle not found")
       except AssertionError as e:
           print(e)


if __name__ == "__main__":
   my_moto = VehicleFactory.create_object(
       "Motorcycle", "Enduro", 250
   )
   print(my_moto.get_specs())
   my_truck = VehicleFactory.create_object("Truck")
   print(my_truck.get_specs())
```

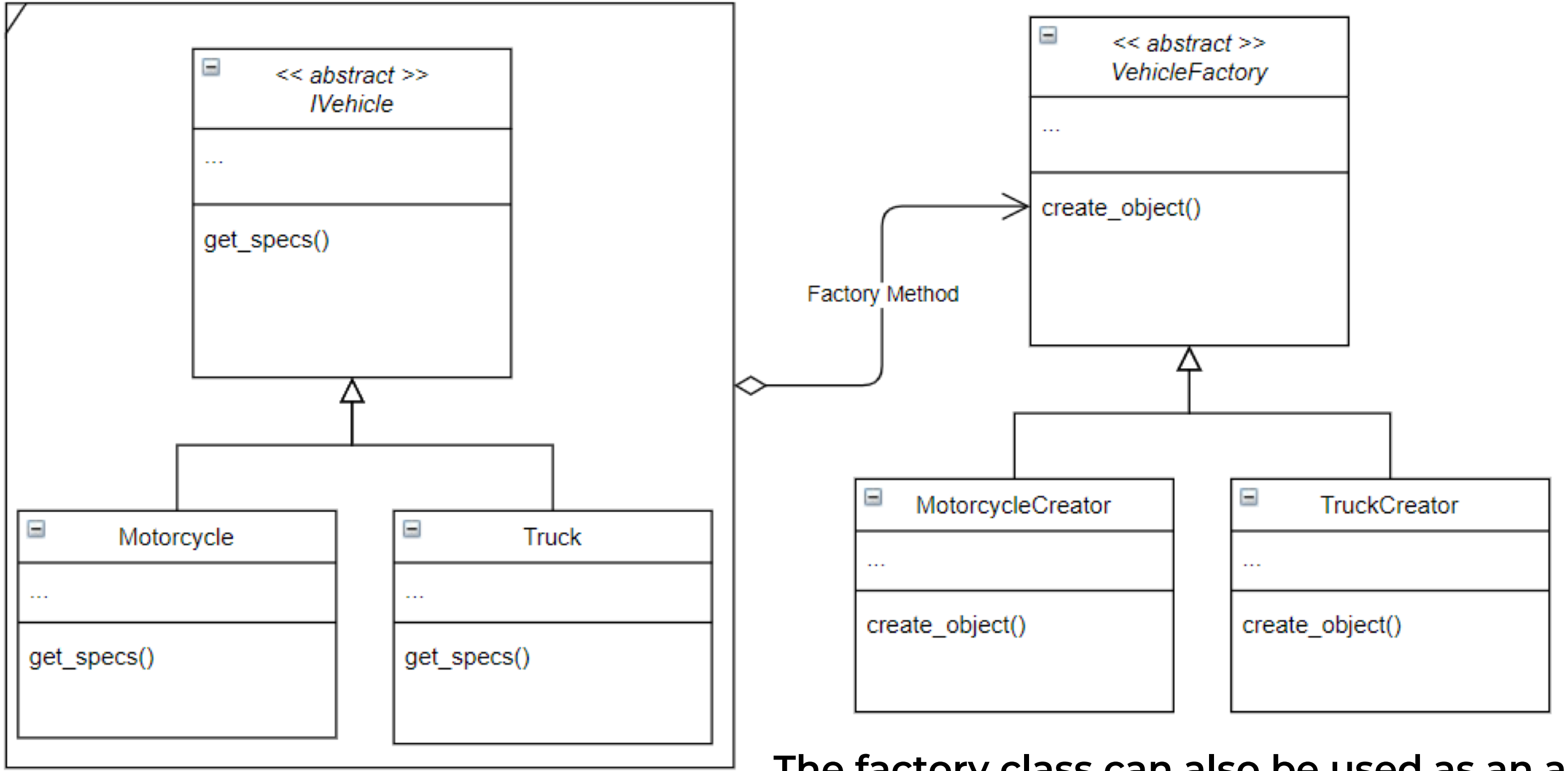
Factory class

This class contains the static method `create_object()` that allows to instantiate objects of the classes defined above.

Runtime

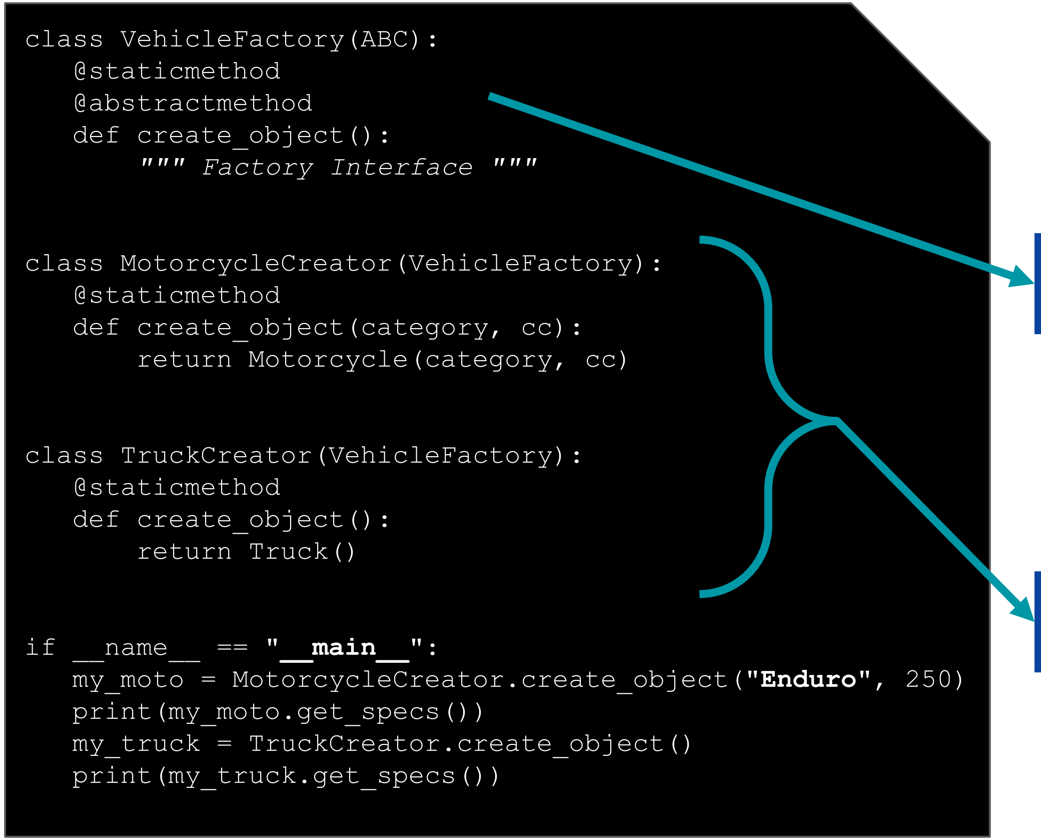
The client decides which objects to instantiate. This can be combined with user input.

Factory Abstract Class



The factory class can also be used as an abstract class.

Factory Abstract Class



```
class VehicleFactory(ABC):
    @staticmethod
    @abstractmethod
    def create_object():
        """ Factory Interface """


class MotorcycleCreator(VehicleFactory):
    @staticmethod
    def create_object(category, cc):
        return Motorcycle(category, cc)


class TruckCreator(VehicleFactory):
    @staticmethod
    def create_object():
        return Truck()


if __name__ == "__main__":
    my_moto = MotorcycleCreator.create_object("Enduro", 250)
    print(my_moto.get_specs())
    my_truck = TruckCreator.create_object()
    print(my_truck.get_specs())
```

- The previous example can be rewritten using creator classes.
- This method respects better the **single responsibility principle (SRP).**

Factory Abstract Class

The factory class can be defined as an abstract class and the method `create_object` is then defined as a static abstract method.

Creator classes

Subclasses of the factory abstract class that will allow to instantiate objects of specific classes.

We learned ...

- That **singletons** are classes that allow to instantiate only one object.
- That the **factory method** is a creational design pattern that allows to create objects.
- That one of the advantages of the **factory method** is the ability to define classes on runtime.

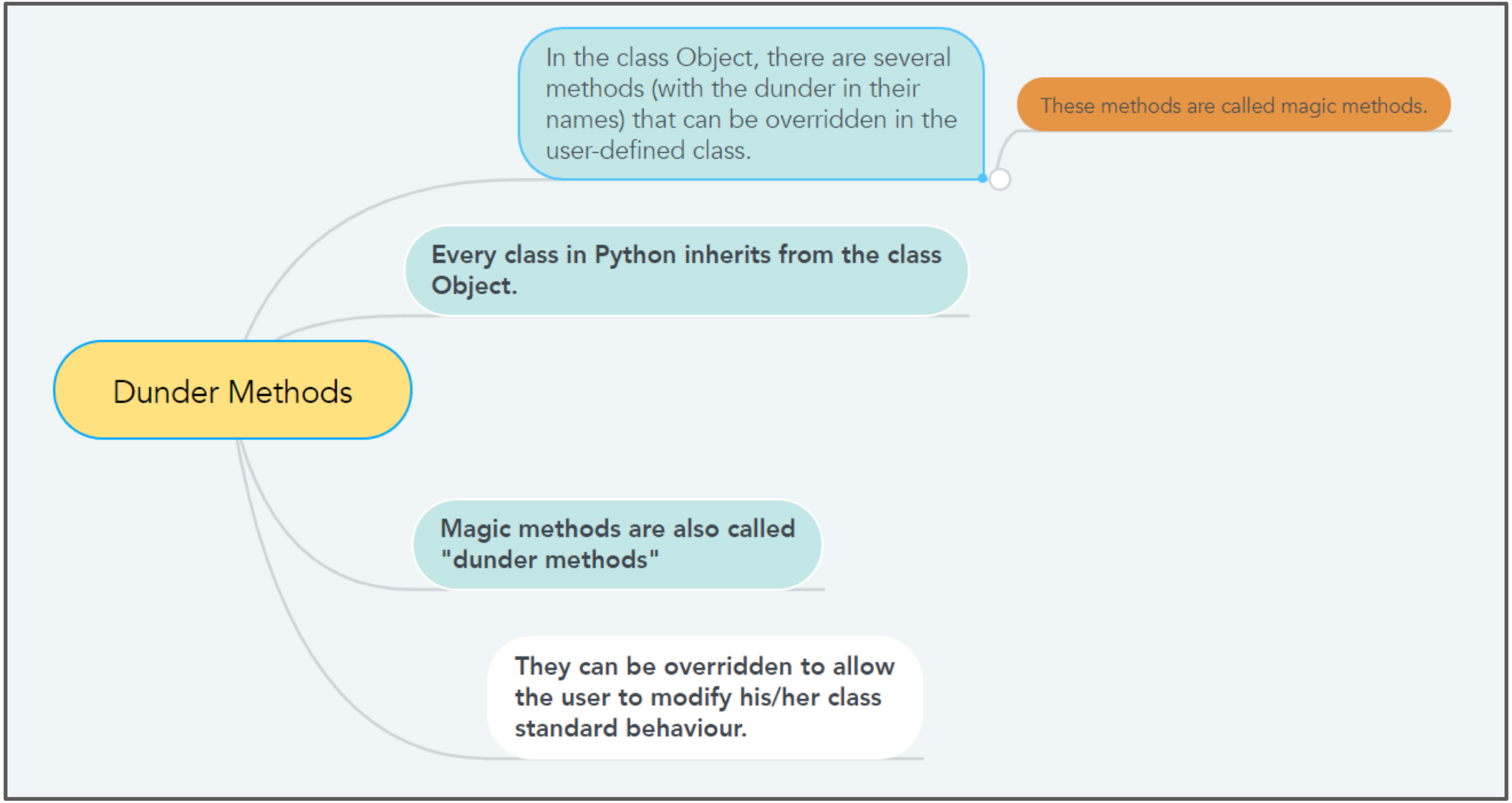
Dunder and Magic Methods

Digital Career Institute

Dunder and Magic Methods

- Dunder means “double underscore” or __
- In the method `__init__()` of the custom classes, the word init is preceded and followed by dunders.
- The methods `__init__()`, `__str__()` and other methods with names starting and ending with dunders are called **magic methods**.
- Most of the operator overloading methods are **magic methods**.

Dunder and Magic Methods



Dunder Methods

In the class Object, there are several methods (with the dunder in their names) that can be overridden in the user-defined class.

These methods are called magic methods.

Every class in Python inherits from the class Object.

Magic methods are also called "dunder methods"

They can be overridden to allow the user to modify his/her class standard behaviour.

Examples of Magic Methods

Magic Method	Functionality
__new__()	This method is called automatically when an object is instantiated. It returns a new object, and then calls the __init__() method. We have seen a use of this in the singleton part.
__str__()	The string representation of the objects of the class. i.e. What will be shown when the print function is used on the object.
__del__()	Destructor method.
__int__(self)	To get called by built-int int() method to convert the class object to an int.

- You can find an expansive list of magic methods in this link: https://www.tutorialsteacher.com/python/magic-methods-in-python

We learned ...

- That **magic methods** are methods provided by default when defining a new class.
- That they can be recognized because their name starts and ends with a **dunder __**.
- That these methods allow the developer to control the default behaviour of the custom defined classes.

Documentation

Resources

- Factory method: https://medium.com/design-patterns-in-python/factory-pattern-in-python-2f7e1ca45d3e

- Magic Methods: https://www.tutorialsteacher.com/python/magic-methods-in-python

- abc module: https://pymotw.com/2/abc/index.html

- Polymorphism: https://www.programiz.com/python-programming/polymorphism

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute