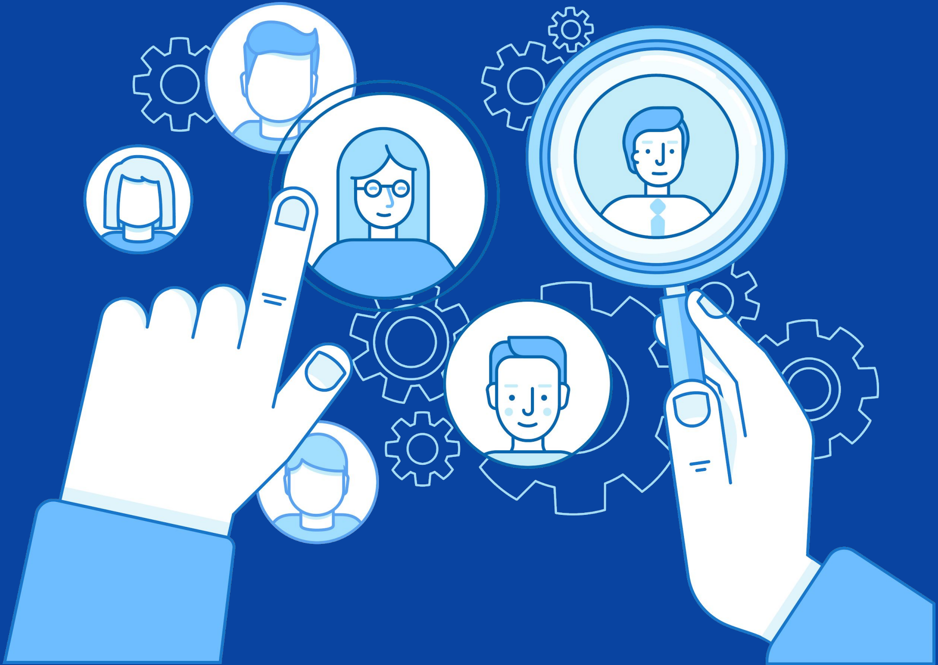
Digital Career Institute

Python Course - OOP in Practice



Digital Career Institute

DCI

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 abstraction paradigm.
- Abstract base classes in Python and mixins.
- The singleton pattern.
- The factory pattern.
- Overloading and polymorphism.
- Type conversion in Python.
- Magic methods.

Digital Career Institute

Topics

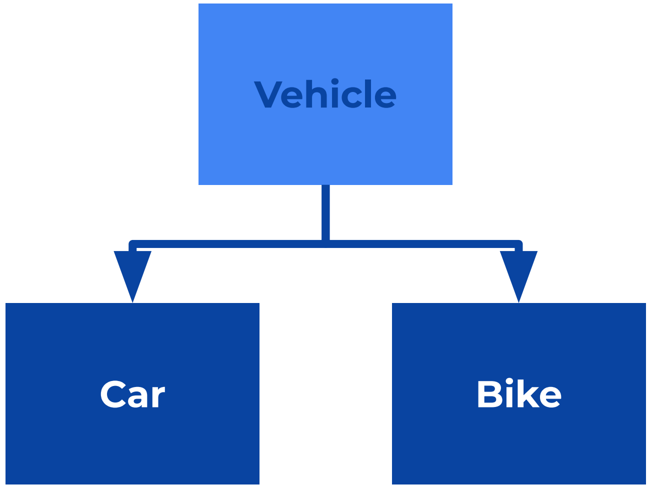
- Abstract classes
- Abstract methods
- Mixins
- The abc Python module
- Singleton pattern
- Factory design method
- Overloading and polymorphism
- Type conversion
- Dunder and magic methods

Abstract Classes

Digital Career Institute

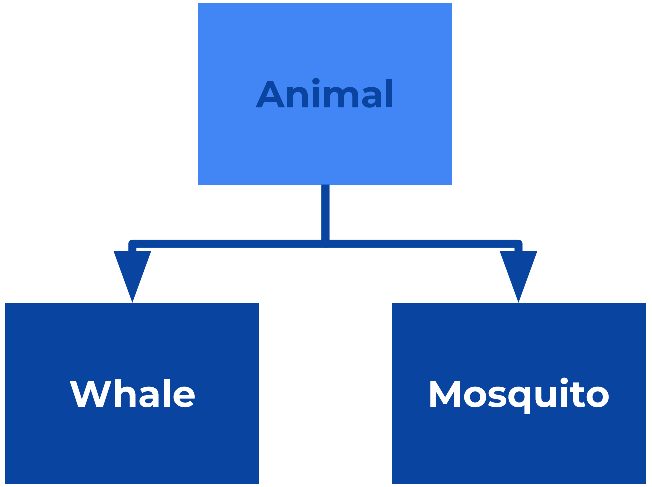
Abstraction in OOP

- So far, it has been mentioned that a class is a pattern from which objects are instantiated.
- This is not always true.
- Sometimes, it can be desirable to define a class that only serves as a base class for other classes to extend.



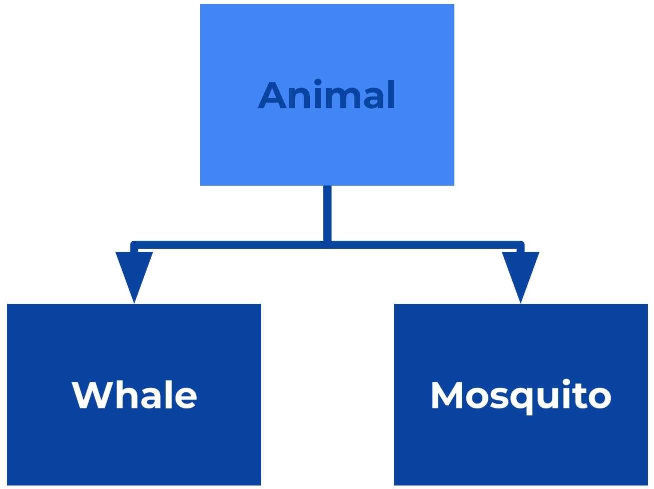
Abstract Classes

- It is not practical to use a class `Animal` to instantiate whales and mosquitoes.
- As animals, they both share common traits (life expectancy, ingestion, excretion,...).
- But defining a whale using the `Animal` class requires having to manually indicate all common traits of whales every time.



Abstract Classes

- **Abstract classes** are only meant to be extended and not instantiated.
- They are used to define some properties and methods that are common to a series of **subclasses or child classes**.



Abstract Methods

- Abstract classes are used to define properties and methods.
- Abstract methods do not offer an implementation. They are defined but they don't define what the method does. The keyword `pass` is often used to define a block that does nothing.

```
>>> class Animal:
...     def reproduce(self):
...         pass
...
>>> class Whale(Animal):
...     def reproduce(self):
...         return Whale()
...
```

Abstract Classes in Python

- Abstract classes are not built–in in the Python core.
- They may be designed as abstract classes (never instantiated), but the language does not treat them any differently.
- In this example, an instance of the abstract class can be created, just like a whale is created.

```
>>> class Animal:
...     def reproduce(self):
...         pass
...
>>> class Whale(Animal):
...     def reproduce(self):
...         return Whale()
...
>>> willy = Animal()
>>> print(willy.reproduce())
None
```

Abstract Base Classes

But abstract classes can be defined as such using the built-in Python module `abc` (Abstract Base Classes).

Abstract methods must use the `@abstractmethod` decorator.

An abstract class defined this way can never be used to instantiate objects.

```
>>> from abc import ABC, abstractmethod
>>> class Animal(ABC):
...     @abstractmethod
...     def reproduce(self):
...         pass
...
>>> willy = Animal()
TypeError: Can't instantiate abstract
class Animal with abstract method
reproduce
```

Abstract Base Classes: Methods

Subclasses created from the abstract class must provide an implementation for each abstractmethod.

If they do not provide it, objects cannot be instantiated from them.

```
>>> from abc import ABC, abstractmethod
>>> class Animal(ABC):
...     @abstractmethod
...     def reproduce(self):
...         pass
...
>>> class Whale(Animal):
...     pass
...
>>> willy = Whale()
TypeError: Can't instantiate abstract
class Whale with abstract method
reproduce
```

Abstract Base Classes: Methods

Subclasses that provide an implementation for all abstract methods can be used normally.

This is useful to detect bugs earlier, as the code explicitly indicates what the problem is when there is one.

```
>>> from abc import ABC, abstractmethod
>>> class Animal(ABC):
...     @abstractmethod
...     def reproduce(self):
...         pass
...
>>> class Whale(Animal):
...     def reproduce(self):
...         return Whale()
...
>>> willy = Whale()
>>> print(willy)
<__main__.Whale object at 0x7f4363f5aa60>
```

Abstract Base Classes: Methods

Abstract methods can be used to group any set of instructions common to all subclasses that need to be executed every time the method is called.

The **`super()`** constructor can be used to execute the instructions in the abstract method.

```
>>> from abc import ABC, abstractmethod
>>> class Animal(ABC):
...     @abstractmethod
...     def reproduce(self):
...         print("Good news!")
...
>>> class Whale(Animal):
...     def reproduce(self):
...         super().reproduce()
...         return Whale()
...
>>> willy = Whale()
>>> print(willy.reproduce())
Good news!
<__main__.Whale object at 0x7f4363f5aa60>
```

Abstract Base Classes: Methods

```
from abc import ABC, abstractmethod
from time import time as tt, ctime as ct


class Vehicle(ABC):
   @abstractmethod
   def do(self, action):
       print(f"Start of action: {action}
at {ct(tt())}")


class Bicycle(Vehicle):
   def do(self, action):
       print(f"{action} the bike
peacefully in the park")


class Car(Vehicle):
   def do(self, action, distance):
       super().do(action)
       print(f"{action} the car for
{distance} km")
```

- Abstract methods can have parameters.
- When overridden in the subclasses, the methods must implement those parameters defined in the abstract method and can have more parameters.

Code parts

The abstract method `do(action)` is redefined in the children classes.

The code in the abstract method is reused.

Abstract Base Classes: Properties

Class properties can also be defined in the same way and their implementation will also be required when subclassing.

They are defined using both the **`@abstractmethod`** and **`@property`** decorators.

The **`@abstractmethod`** decorator must always be the closest to the method definition.

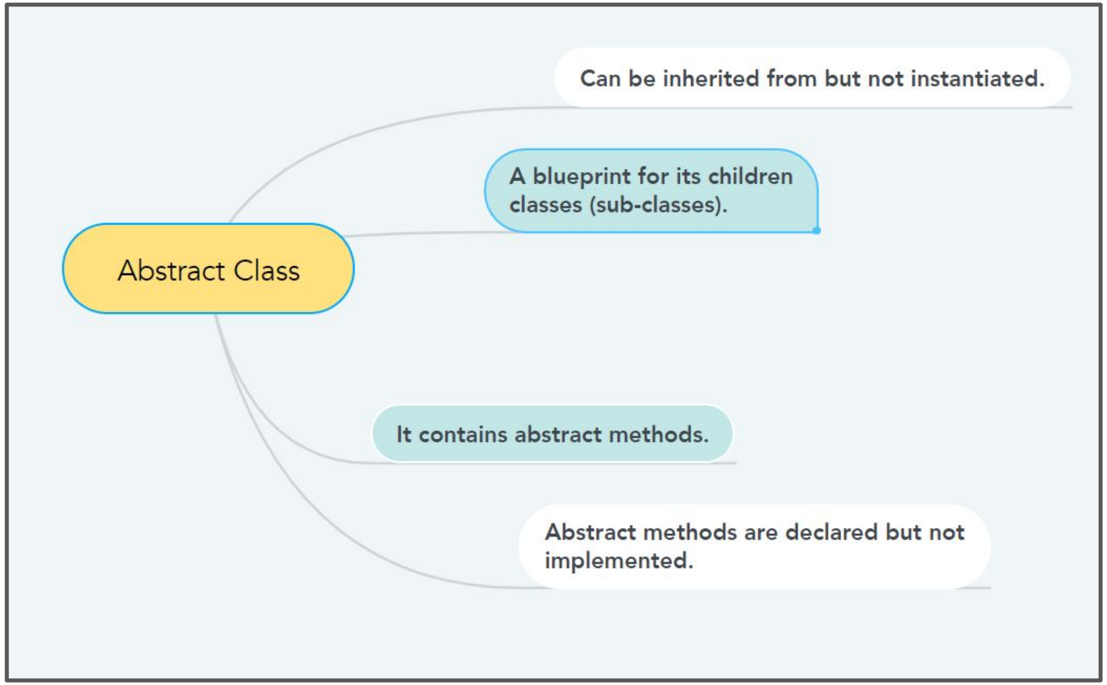
```
>>> from abc import ABC, abstractmethod
>>> class Animal(ABC):
...     @property
...     @abstractmethod
...     def reproduction(self):
...         pass
...
>>> class Whale(Animal):
...     pass
...
>>> willy = Whale()
TypeError: Can't instantiate abstract
class Whale with abstract method
reproduction
```

Abstract Base Classes: Properties

If the subclass has the required properties, no error appears and the class can be used normally to instantiate objects.

```
>>> from abc import ABC, abstractmethod
>>> class Animal(ABC):
...         @property
...         @abstractmethod
...         def reproduction(self):
...             pass
...
>>> class Whale(Animal):
...         reproduction = "sexual"
...
>>> willy = Whale()
>>> print(willy.reproduction)
sexual
```

Abstract Base Classes - Summary


Abstract Class
Can be inherited from but not instantiated.
A blueprint for its children classes (sub-classes).
It contains abstract methods.
Abstract methods are declared but not implemented.

OOP Design Patterns

Digital Career Institute

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

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
- …

Mixins

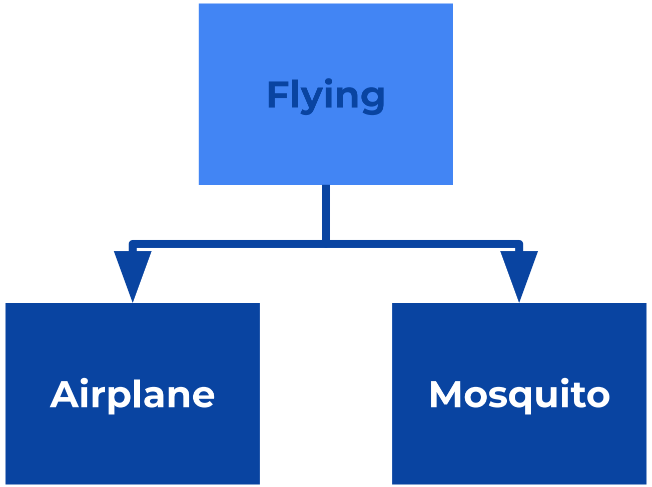
Digital Career Institute

Class Mixins

Mixins are similar to abstract classes and are defined and implemented using a special design pattern.

Like abstract classes, they are not meant to be directly instantiated into objects, but used by the subclasses.

They are used to **mix** a specific feature **in** otherwise unrelated classes.

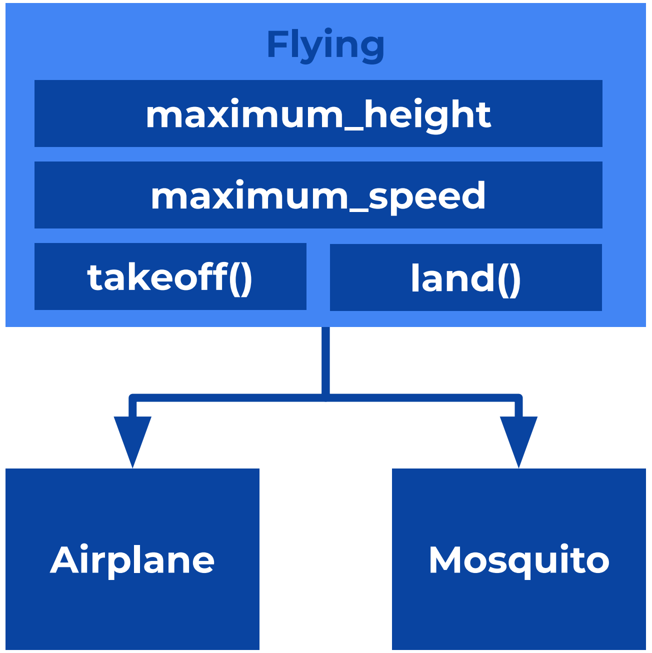


Class Mixins

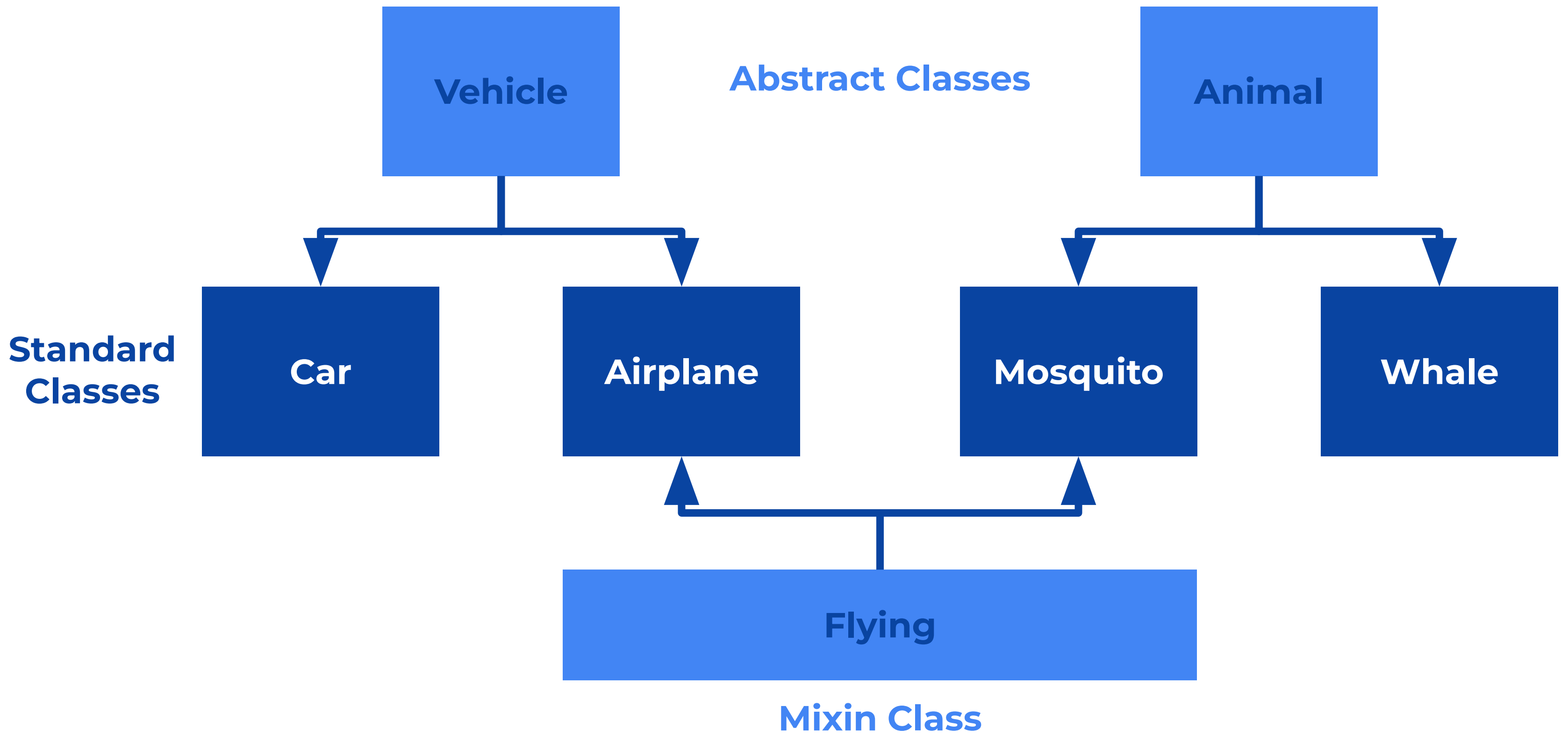
Mixins define and implement a **single, well-defined feature**.

As opposed to abstract classes, they provide a default implementation of the properties and methods involved in this feature.

Both airplanes and mosquitoes will have a `takeoff` and a `land` method, because they both fly.



Class Mixins

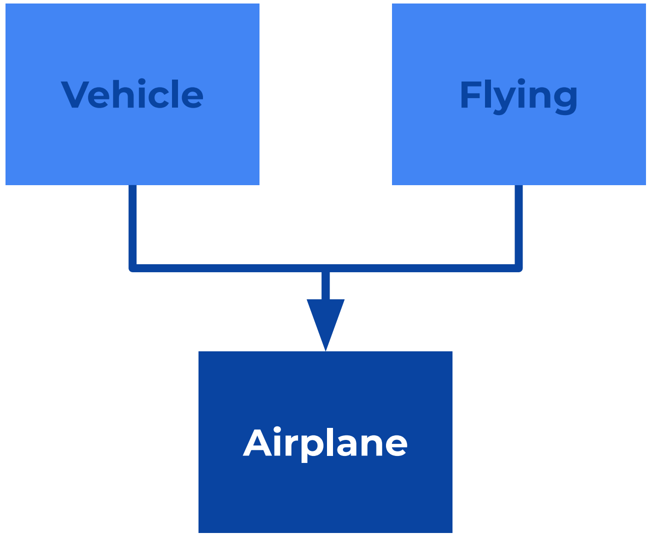


Python Mixins: Multiple Inheritance

In Python, mixins are often used in cases of multiple inheritance.

An airplane is both a vehicle and a flying entity.

In Python, the class `Airplane` will extend both the abstract class `Vehicle` and the mixin `Flying`.



Python Mixins

Python does not implement a specific type for mixins.

It is the programmer's responsibility to design the class as a proper mixin.

Very often, the mixin class is named adding the `Mixin` suffix, so that it is clear the purpose of the class.

```
>>> class Vehicle(ABC):
...       # instructions
...
>>> class Animal(ABC):
...       # instructions
...
>>> class FlyingMixin:
...       # instructions
...
>>> class Airplane(Vehicle, FlyingMixin):
...       # instructions
...
>>> class Mosquito(Animal, FlyingMixin):
...       # instructions
...
```

Python Mixins

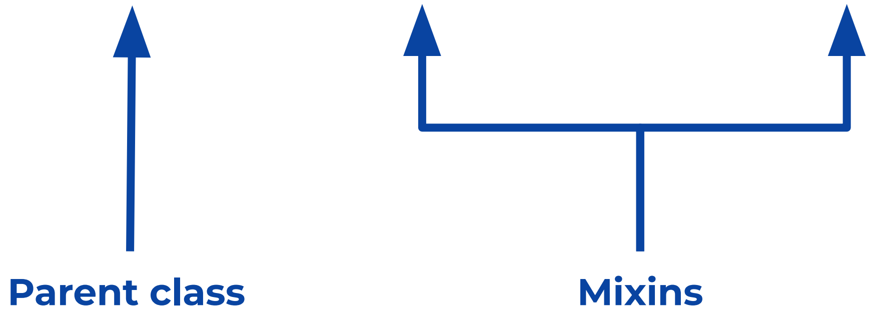
Multiple mixins can be used to extend a class.

This way, an airplane will have the method `takeoff` but not the method `bite`. Spiders will have the method `bite` but not the `takeoff` method, and mosquitoes will have both methods.

```
>>> class Airplane(Vehicle, FlyingMixin):
...     # instructions
...
>>> class Spider(Animal, BitingMixin):
...     # instructions
...
>>> class Mosquito(Animal, FlyingMixin
...                  BitingMixin):
...     # instructions
...
```

Python Mixins

```
class Mosquito(Animal, FlyingMixin, BitingMixin):
```



First, all the features of the parent class are loaded,
then each of the mixins is merged into it
in order from left to right.

Mixins & Interfaces

In computer science an **interface** is often similar to a **mixin**. Interfaces are also meant to provide the definition of a feature.

As opposed to mixins, interfaces don’t implement the declared methods and it is the child classes who must do so.

In most programming languages, interfaces cannot be used with multiple inheritance.

Python does not have a type implementation for interfaces.

We learned …

- That abstract classes are declared and extended but never instantiated.
- That abstract classes are used to group a set of features that are common for all the subclasses.
- That the `abc` module of Python is used to enforce the correct behavior of abstract classes.
- That there are different design patterns in OOP.
- That mixins are used to extend classes with some well-defined and specific features that are not specific to those classes or their parent classes.

Singletons

Digital Career Institute

What is a Singleton?

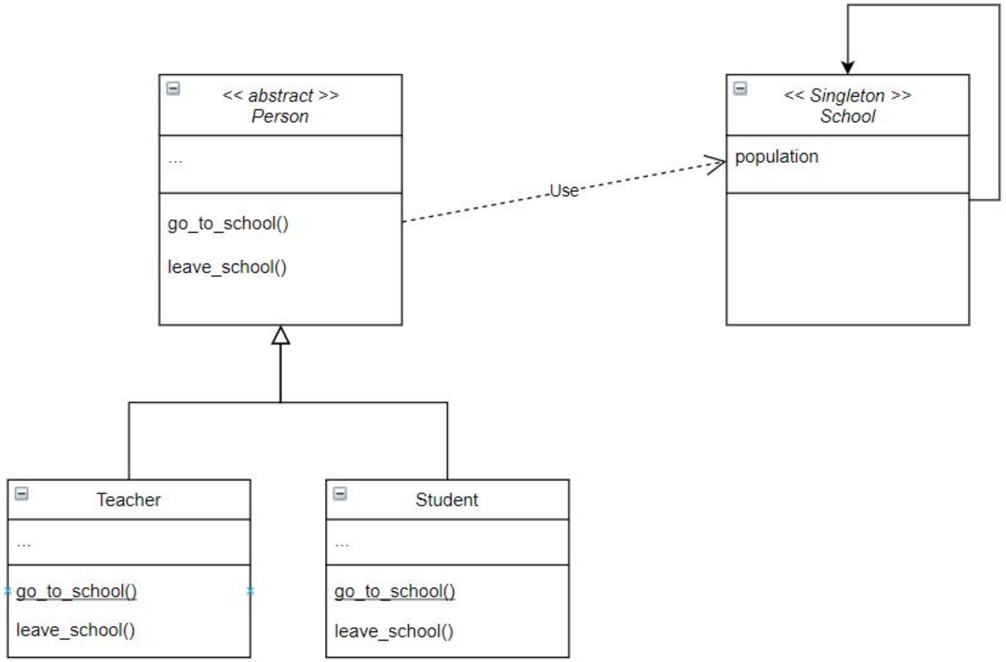
- A singleton pattern is a class where only one instance is ever created.
- Because of this, they are often used to provide the same object to any part of the code.
- This means that they act as a global scope manager.
- There are different ways to implement a singleton.

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
```

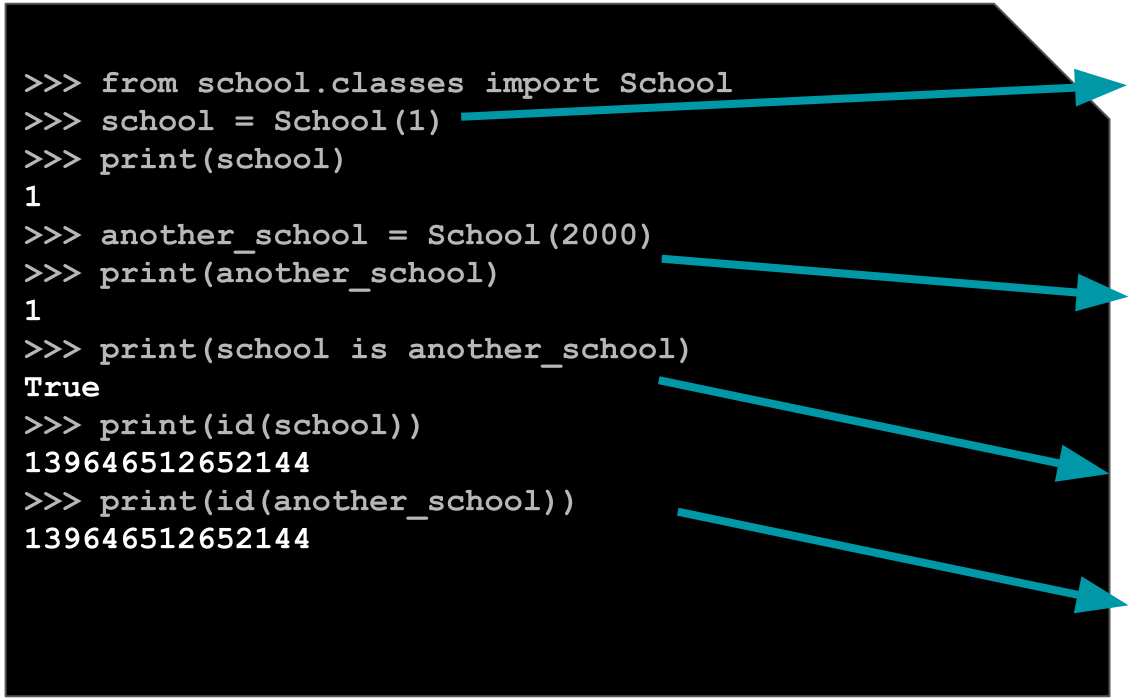
The **__Singleton** class contains the logic of our target class.

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.

Singleton Decorator

school/classes.py

```
def singleton(class_):
   __instances = {}

   def get_instance(*args, **kwargs):
       if class_ not in __instances:
           __instances[class_] = class_(*args, **kwargs)
       return __instances[class_]
   return get_instance

@singleton
class FirstClass:
   def __init__(self, m):
       self.val = m

@singleton
class SecondClass:
   def __init__(self):
       self.val = 0
```

A decorator can be defined to convert any class into a singleton.

```
>>> a = FirstClass(1)
>>> b = FirstClass(23)
>>> print(b.val)
1
>>> print(id(another_school))
139646512652144
```

Singleton Use Cases

Singletons are not often used but they may be useful in certain situations, like:

- **Configuration data**. We may have a single set of configuration parameters and we may want to store them in a class.

- **Shopping cart**. On the back-end we may need to instantiate various shopping carts, for each user. But on the front-end, each user often has a single shopping cart.

Notes and Remarks

- Singletons are a bit controversial as they violate the **single responsibility principle (SRP)**.

- They are not recommended for testability reasons. See https://www.youtube.com/watch?v=acjvKJiOvXw.

- There are other ways to implement singletons. Notably:

    - Using a metaclass.
    - Using a shared_state (for more: http://www.aleax.it/Python/5ep.html).

- In this course it’s advised to use decorators as described in the previous slide. It’s a simple, yet elegant and efficient, way to implement singletons.

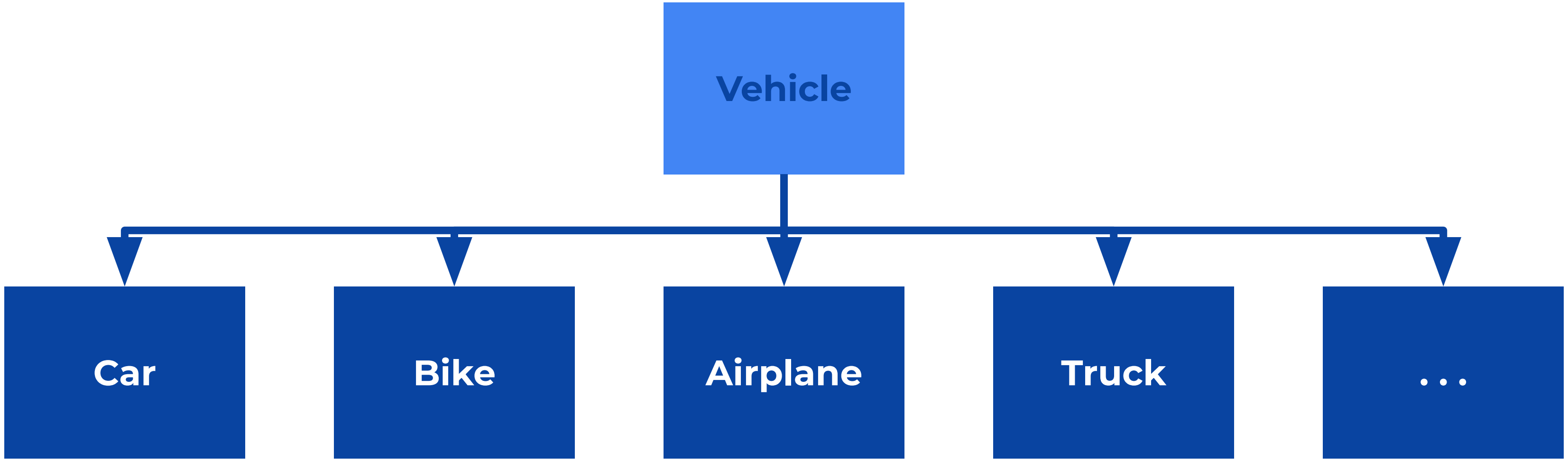
OOP Design Patterns: Factories

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

```
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

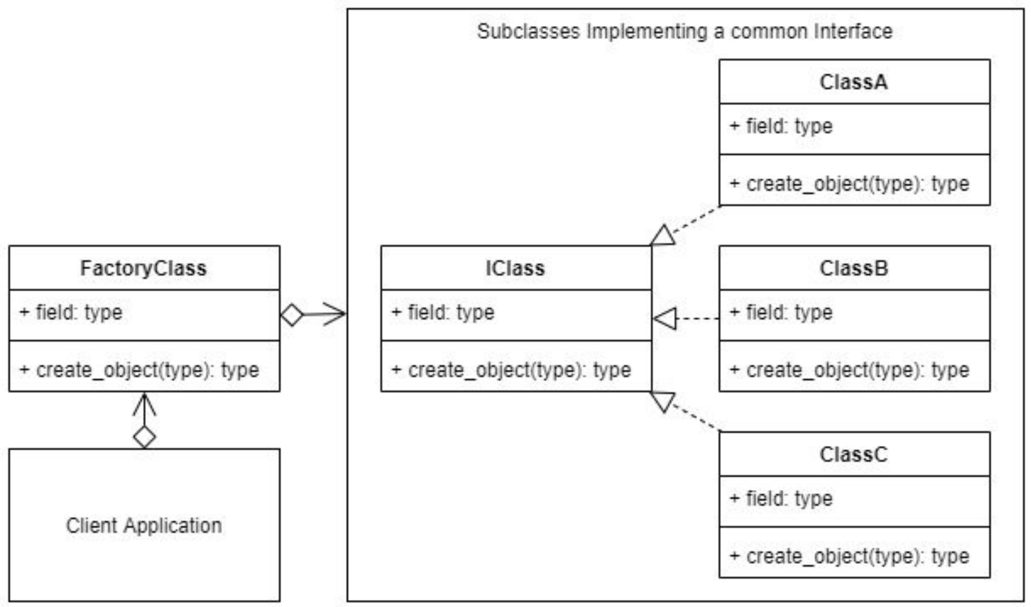
```python
def factory(type):
    if type == "Car":
        return Car
    if type == "Bike":
        return Motorbike
    if type == "Airplane":
        return Airplane
    if type == "Whale":
        return Whale
```

classes.py

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

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

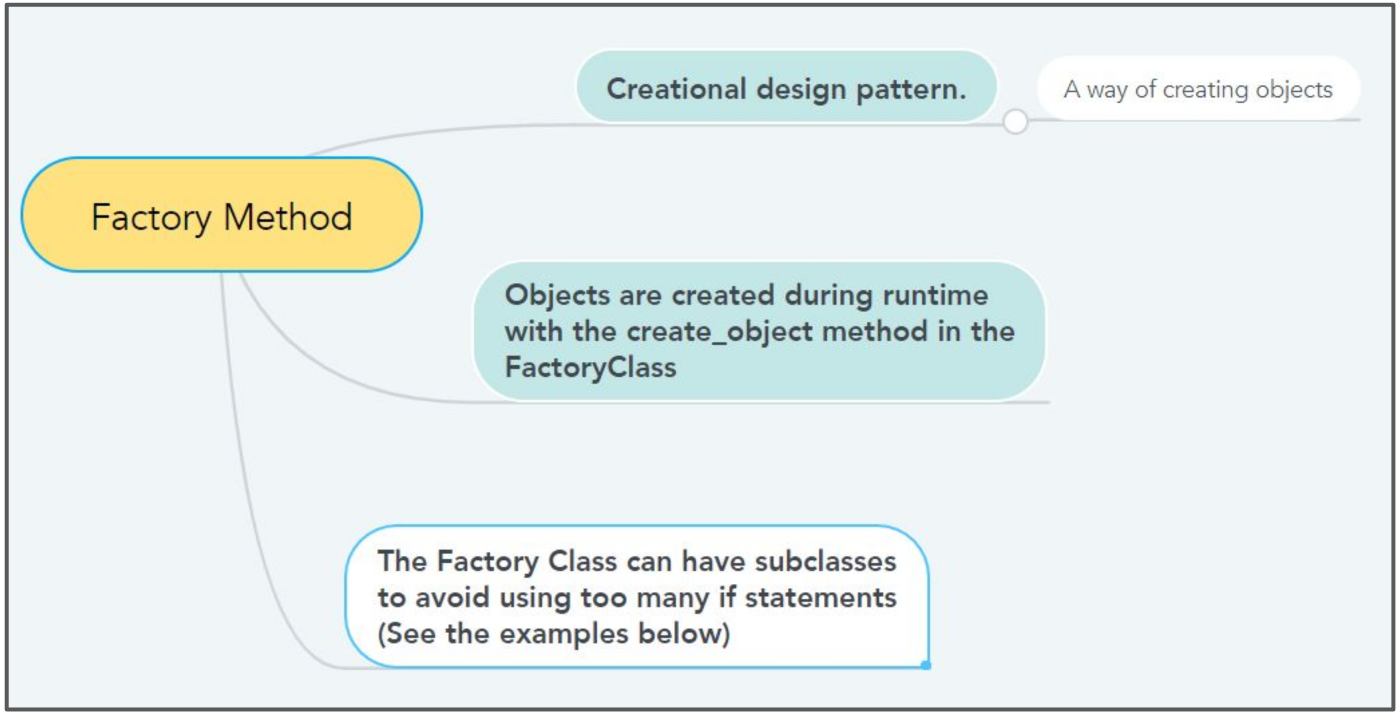
Factory Method - Explanation


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

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

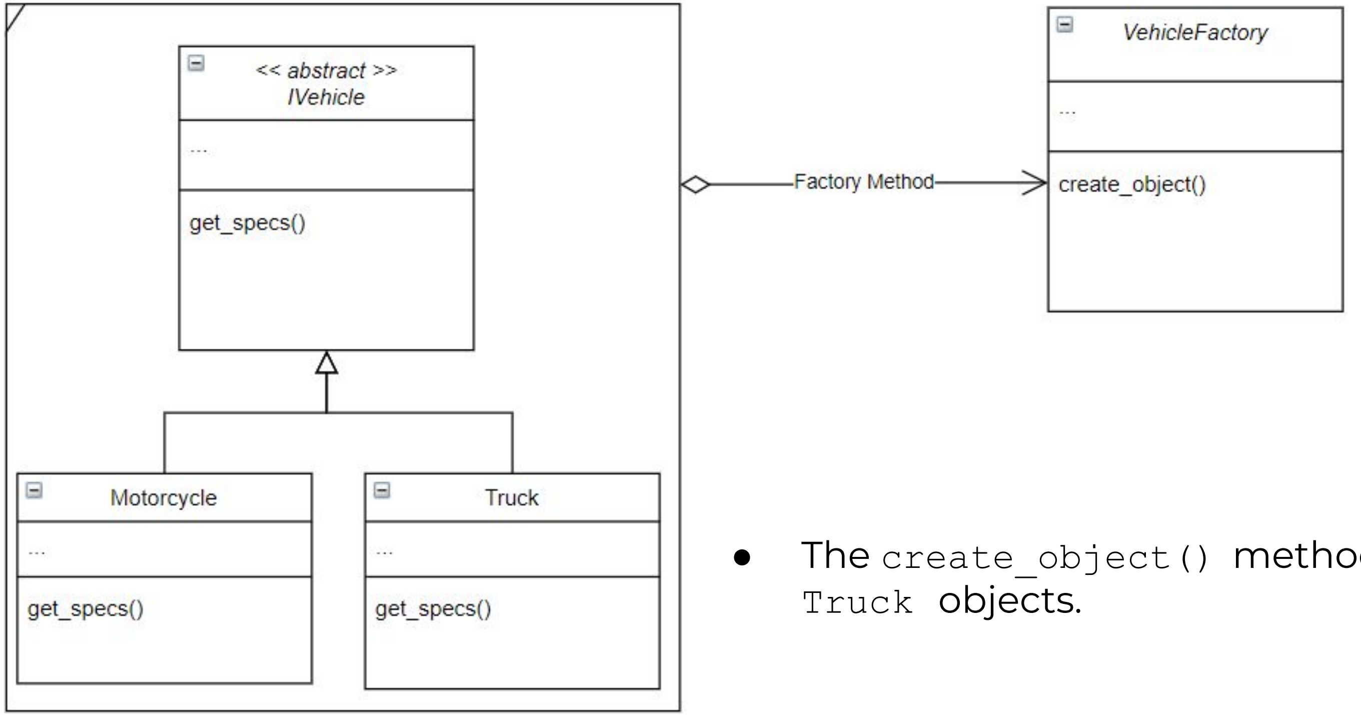
- `IClass` is an abstract class.
- `ClassA`, `ClassB` and `ClassC` are its subclasses.
- The `FactoryClass` contains the method `create_object()`.
- During runtime, and depending on the client choices, the `FactoryClass` creates an object of `ClassA`, `ClassB` or `ClassC`.

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())
```

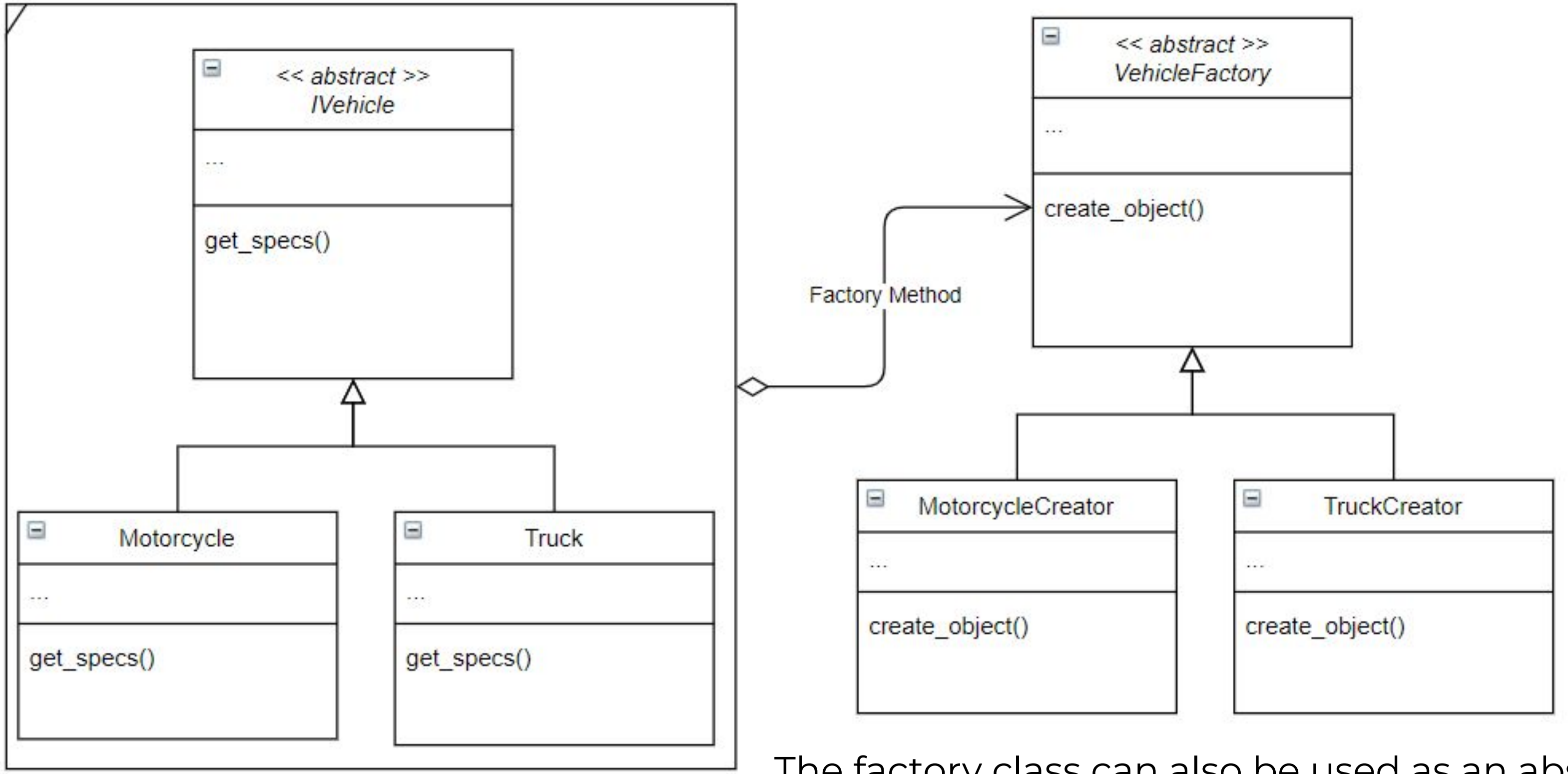
Factory class

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.

Overloading

What is Overloading?

Overloading is the mechanism by which a single object can be treated differently depending on how it is being used.

There are two types of overloading:

Function Overloading

Operator Overloading

A function can do different things depending on the arguments given.

Different operators may do different things depending on the type of the operands.

Method Overloading

One same action may sometimes be done in different ways depending on the arguments available.

This, sometimes, means creating multiple functions named different, requiring the developer to remember all of them.

Method overloading allows for a more compact and clear way of dealing with these situations.

classes.py

```python
class Point:
    def setFromLatLng(self, lat, lng):
        self.lat = lat
        self.lng = lng
    def setFromList(self, coords):
        self.lat = coords[0]
        self.lng = coords[1]
    def setFromDict(self, coords):
        self.lat = coords["lat"]
        self.lng = coords["lng"]
    def set3DFromLatLng(self, lat, lng, alt):
        self.lat = lat
        self.lng = lng
        self.alt = alt
    def set3DFromList(self, coords):
        self.lat = coords[0]
        self.lng = coords[1]
        self.alt = coords[2]
    def ...
```

Method Overloading

There is no built-in solution in the Python core library to manage method overloading.

One way of accomplishing this is to set the logic inside the body of the method. Packing arguments is very convenient with this approach.

This way, the developer only needs to call `set`, with any number and types of arguments available.

classes.py

```python
class Point:
    def set(self, *args):
        if len(args) == 1:
            if isinstance(args[0], list):
                self.lat = args[0][0]
                self.lng = args[0][1]
        else:
            self.lat = args[0]
            self.lng = args[1]
```

```
>>> from classes import Point
>>> a = Point(); a.set(2, 42)
>>> print(a.lat, a.lng)
2 42
>>> a.set([3, 41])
>>> print(a.lat, a.lng)
3 41
```

Method Overloading

The `functools` module provides a decorator named `singledispatchmethod` that can be used to define method overloading with a cleaner and more readable code.

```
>>> from classes import Point
>>> a = Point(); a.set(2, 42)
>>> print(a.lat, a.lng)
2 42
>>> a.set([3, 41])
>>> print(a.lat, a.lng)
3 41
```

classes.py

```
from functools import singledispatchmethod


class Point:
    @singledispatchmethod
    def set(self, arg):
        raise NotImplementedError()

    @set.register
    def _(self, lat: float, lng: float):
        self.lat = lat
        self.lng = lng

    @set.register
    def _(self, coords: list):
        self.lat = coords[0]
        self.lng = coords[1]
```

Function Overloading

The same principle applies to custom functions.

If many similar functions are needed it is often better to use overloading so that the code is more readable and less prone to bugs.

The approach is the same as with methods: manually adding the logic in the function or using `functools`. The decorator, this time, is named `singledispatch`.

functions.py

```
def getPointFromLatLng(self, lat, lng):
    return Point(lat, lng)

def getPointFromList(self, coords):
    return Point(coords[0], coords[1])

def getPointFromDict(self, coords):
    return Point(
        coords["lat"], coords["lng"]
    )
```

Built-in Function Overloading

Some built-in Python functions, like `len`, can be overloaded to accept user-defined classes.

These functions do not work if we pass them custom objects, like `line`, but they do if specific methods in these classes are overloaded.

classes.py

```python
class Line(GeometricObject):

    def __init__(self, coordinates):
        self.coordinates = coordinates
```

```
>>> from classes import Line
>>> line = Line([[2, 42], [3, 43], [4, 44]])
>>> print(len(line))
TypeError: object of type 'Line' has no len()
```

Built-in Function Overloading

```
>>> line1 = Line(
...       [[2, 42], [3, 43], [4, 44]]
... )
...
>>> line2 = Line([[5, 55], [6, 66]])

>>> print(len(line1))
3

>>> print(line2)
[[5, 55], [6, 66]]

>>> print(sum([line1, line2]))
270
```

functions.py

```python
class Line(GeometricObject):

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

    def __len__(self):
        return len(self.coordinates)

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

    def __radd__(self, other):
        return other + sum([
            coord[0] + coord[1]
            for coord in self.coordinates
        ])
```

Built-in Function Overloading

- Below is a sample list of operators and the class methods to overload them:

Operator	Class Method
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)
-a	a.__neg__()
str(a)	a.__str__()
abs(a)	a.__abs__()
len(a)	a.__len__(b)
a[k]	a.__getitem__(k)
a[k] = v	a.__setitem__(k, v)

Operator Overloading

- Many operators will also behave differently depending on the type of the operands used.

```
>>> print(1 + 1)
2
>>> print("1" + "1")
11
>>> print([1] + [1])
[1, 1]
```

- These operators can also be used with user-defined classes thanks to operator overloading.

- We can do so by implementing special methods in the user defined class. For example the special method `__add__()` is called when the operator + is used.

Operator Overloading - Example

```python
class Subject:
   def __init__(self, name, factor):
       self.name = name
       self.factor = factor

   def __add__(self, other):
       return self.factor + other.factor

   def __mul__(self, other):
       if isinstance(other, Subject):
           return self.factor * other.factor


s1 = Subject("Maths", 3)
s2 = Subject("Networks", 4)
print(s1 + s2)  # 7
print(s1 * s2)  # 12
```

Overloading operators

User defined methods for overloading of the operators **+** and *****.

The operators **+** and ***** call methods `__add__` and `__mul__` of the class `Subject`.

In the example above, when calling `s1 + s2`, the python interpreter calls the class method `s1.__add__(s2)`

Operator Overloading - Examples

- Below is a sample list of operator and the class methods to overload them:

Operator	Class Method
a + b	a.__add__(b)
a - b	a.__sub__(b)
a * b	a.__mul__(b)
a / b	a.__truediv__(b)
a // b	a.__floordiv__(b)
a % b	a.__mod__(b)
a ** b	a.__pow__(b)

Polymorphism

Digital Career Institute

Polymorphism

The word polymorphism means multiple forms or shapes.

In computer programming it is the ability of a method, a function or an operator to behave differently depending on the context.

Overloading is one way of defining polymorphism and it has to do with the arguments passed to the functions or the operands used in the operators.

```
>>> 1 + 3
4
>>> "ab" + "Fg"
'abFg'

>>> 3 * 5
15
>>> "Hello" * 3
'HelloHelloHello'

v1 = 10
v2 = "Hello Berlin"
v3 = [5, True, "Grass"]
v4 = {"name": "Rafael", "age": 35}

print(v1)  # 10
print(v2)  # Hello Berlin
print(v3)  # [5, True, 'Grass']
print(v4)  # {'name': 'Rafael', 'age': 35}
```

Polymorphic Classes

Another way of polymorphism are **Polymorphic Classes** and they involve *class inheritance* and *method overriding*.

Different subclasses of the same parent class may provide different implementations of the same method.

This way, the same interface `travel` can be used to operate on different types of objects and do different things (`gallop` or `slither`) depending on the type and without the developer having to manually code this logic on the main script.

```
from abc import ABC, abstractmethod

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

class Snake(Animal):
   def move(self):
       return "Snake Slithers"

class Horse(Animal):
   def move(self):
       return "Horse Gallops"

def travel(obj, point_a, point_b):
   print(obj.move(), "From", point_a, "To", point_b)


travel(Horse(), "Forest", "Desert")
# >>> Horse Gallops From Forest To Desert
travel(Snake(), "Hole", "Top of the rock")
# >>> Snake Slithers From Hole To Top of the rock
```

Polymorphism in OOP and Python

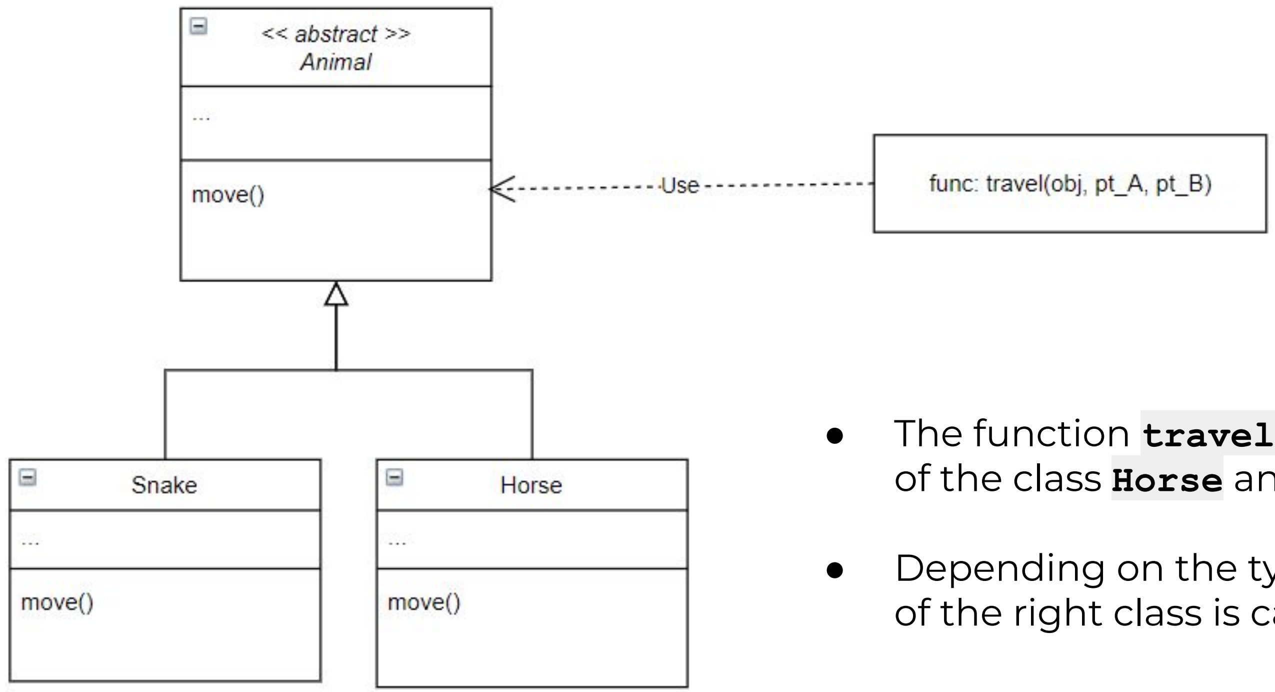
```
from abc import ABC, abstractmethod

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

class Snake(Animal):
   def move(self):
       return "Snake Slither"

class Horse(Animal):
   def move(self):
       return "Horse Gallop"

def travel(obj, point_a, point_b):
   print(obj.move(), "From", point_a, "To", point_b)


travel(Horse(), "Forest", "Desert")
# >>> Horse Gallop From Forest To Desert
travel(Snake(), "Hole", "Top of the rock")
# >>> Snake Slither From Hole To Top of the rock
```

Polymorphism in OOP

Abstract class **`Animal`** and its subclasses **`Snake`** and **`Horse`**. The abstract method **`move()`** is implemented in both subclasses.

The function **`travel`** is defined to deal with any type that implements a **`move()`** method.

Polymorphic Classes



- The function **`travel()`** uses the method **`move()`** of the class **`Horse`** and **`Snake`**.
- Depending on the type of obj, the right method of the right class is called.

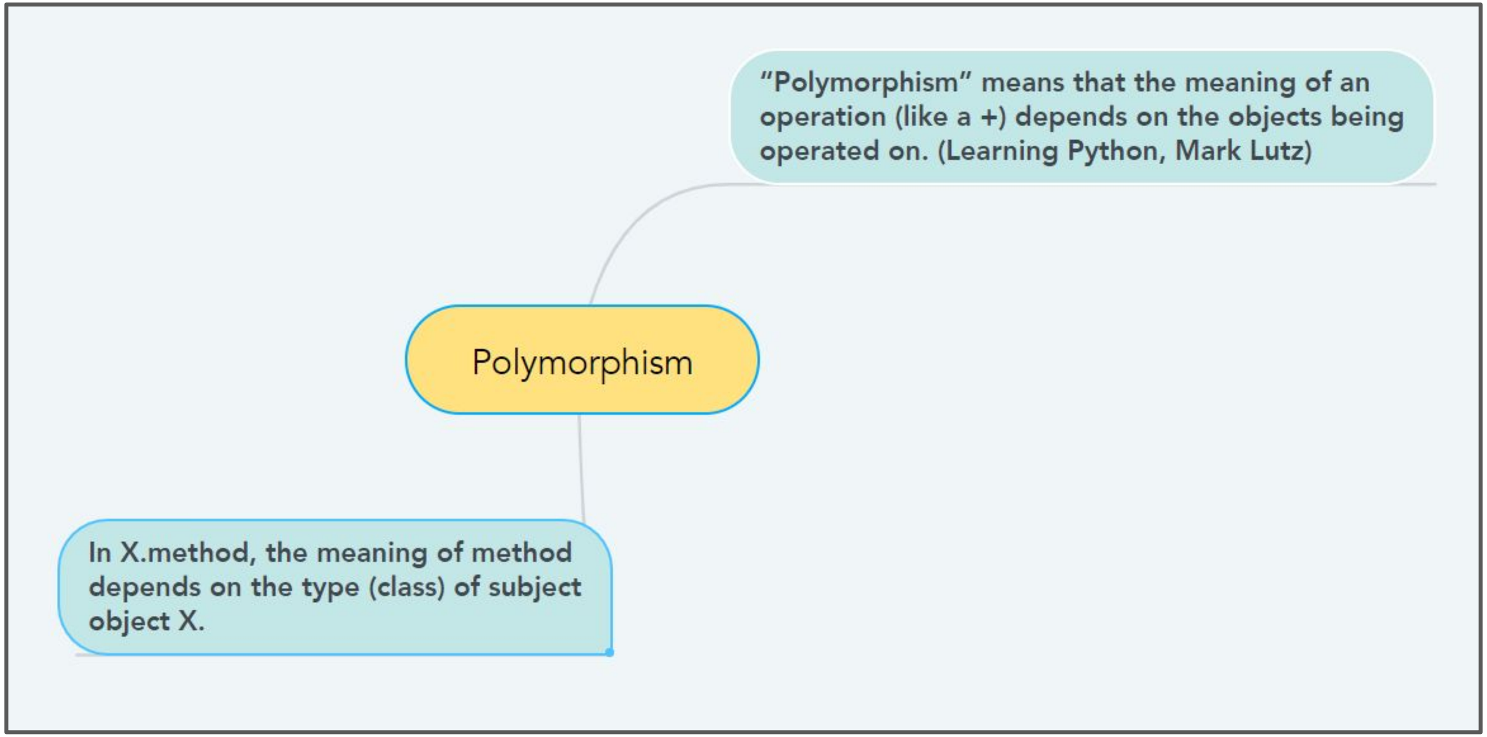
Polymorphism in Python

Since Python is a ‘dynamic types’ language, Polymorphism is rampant in it. This represent the one the strengths of this language.

As long as the object type has an interface (protocol) that fits the function or operator applied, the instruction will run without error.

Even more powerful; The polymorphism allows multi-type in function and operations e.g. **print(“a” , 5)**, **3 * “more”.**

Conclusion



"Polymorphism" means that the meaning of an operation (like a +) depends on the objects being operated on. (Learning Python, Mark Lutz)

In X.method, the meaning of method depends on the type (class) of subject object X.

We learned ...

- **Overloading** is a type of polymorphism that allows for different types of arguments or operands to change the behavior of a function or operator.

- **Polymorphism** means that the behavior of a method, function or operator depends on the context.

- **Polymorphic Classes** allow for a same method to do different things depending on the class they belong to.

- **Polymorphic classes** use class inheritance and method overriding to achieve this goal.

- **Overloading** and **Polymorphism** are techniques used to provide a cleaner and more readable and bug-free code.

Digital Career Institute

Casting and Conversion

Digital Career Institute

Conversion in Python

- Type casting is basically type conversion => Change a Python object from one type to another.

- There are 2 types of conversion:

  - Implicit type conversion.
  - Explicit type conversion, also called **type casting.**

Implicit Conversion

- Implicit conversion happens when Python converts types without a clear instruction to do so in the code. It's done automatically.

```
a = 1
b = 2
pi = 3.14

print(a/b)
#>>> 0.5

print(b/a)
#>>> 2.0

print(a + pi)
#>>> 4.14

c = "Hello"
d = (1, 2, 3)

print(c + d)
#>>> Error
```

Implicit conversion

Python automatically converted the numbers to float when it was needed.

Python however doesn't automatically convert strings to tuples.
Solution: **Type Casting** (explicit conversion).

Explicit Conversion: Type Casting

- The user indicates the conversion.
- It’s done by calling one of the functions **tuple(), list(), int(), str(), …**

```
c = "Hello"
d = (1, 2, 3)
print(tuple(c) + d)
#>>> ('H', 'e', 'l',
'l', 'o', 1, 2, 3)


m = 120
print(list(m))
#>>> TypeError:
'int' object is not
iterable
```

Type Casting

`tuple(c)` converted `c` from string to tuple.

Not every type casting is possible. Only iterables can be cast into other iterables.

TypeError and ValueError

When casting in Python, two types of errors can happen:

TypeError:

When the conversion is impossible between the two types indicated.

Example:
a = (1, 2, 3)
b = int(a) #>> generates an error.

Because the type `tuple` cannot be cast to `int`.

ValueError:

When the conversion is possible between the two types, but the value is wrong and can't be converted.

Example:
int("34") #>> Works well. It casts string to int.
int("Moi") #>> generates a ValueError.

Because the value `"Moi"` can't be cast to an `int`.

We learned …

- That, most Python objects can be converted from one type to another.
- That implicit conversion is done by Python automatically when needed.
- That explicit conversion (type casting), is done by the user in the code.
- That not all type conversions are possible. `ValueError` or `TypeError` will be generated if the conversion is wrong.

Digital Career Institute

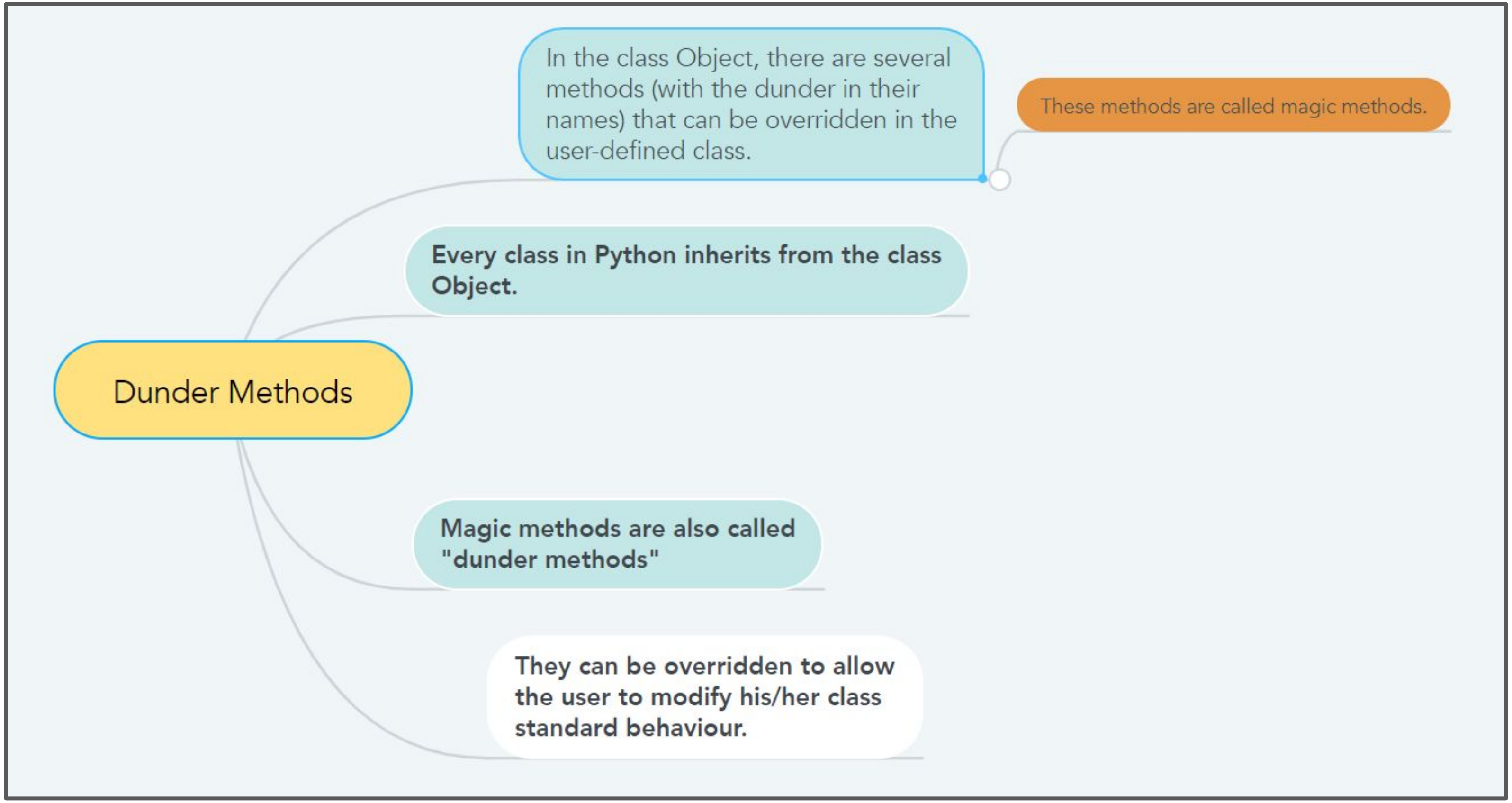
Dunder and Magic Methods

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.

Digital Career Institute

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