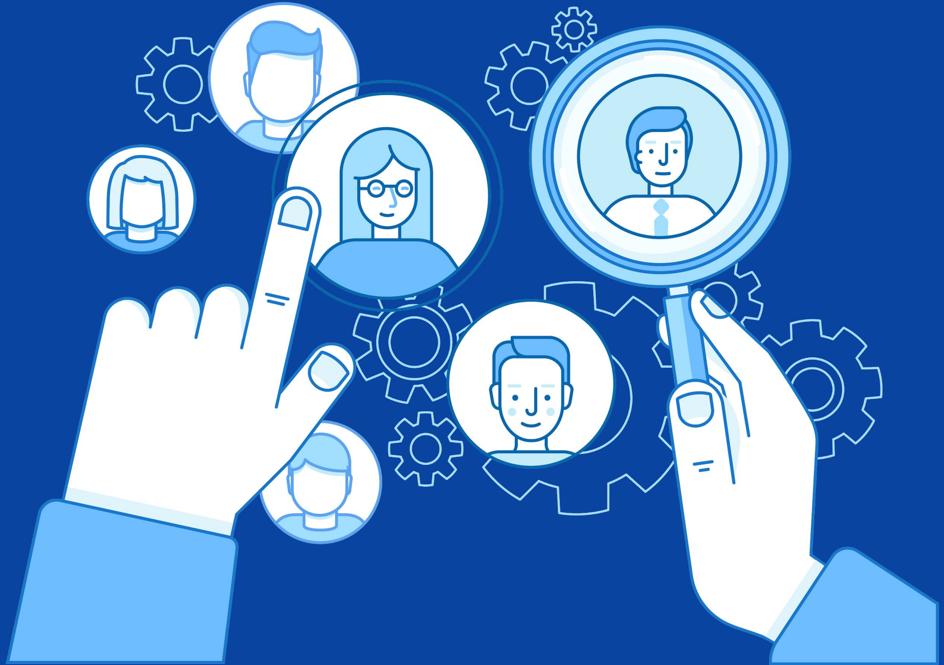
Digital Career Institute

Python Course - OOP in Practice



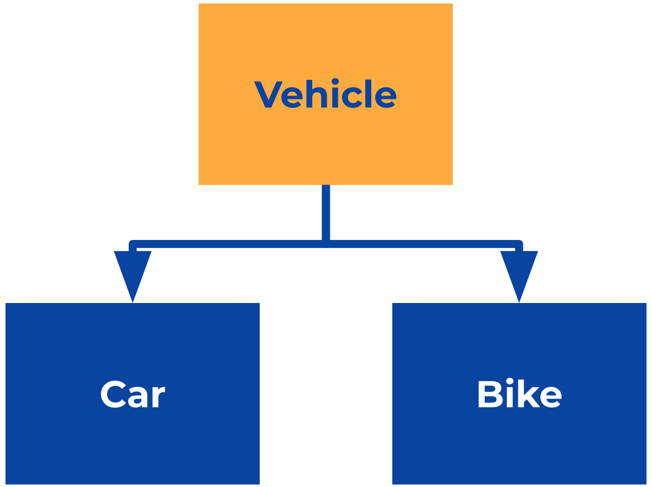
Digital Career Institute

Abstract Classes

Digital Career Institute

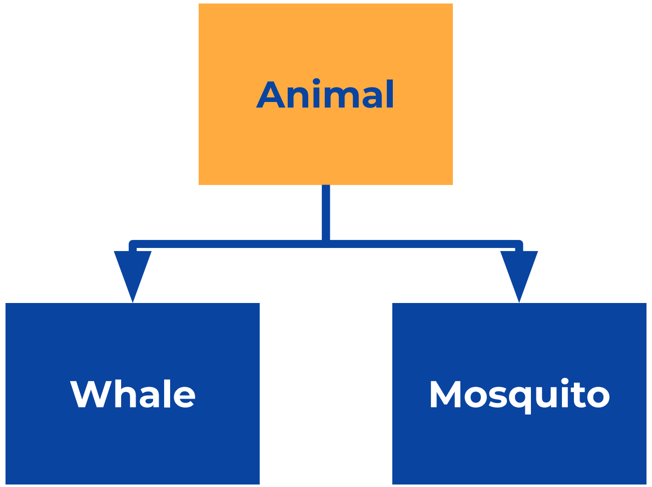
Abstraction in OOP

- 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

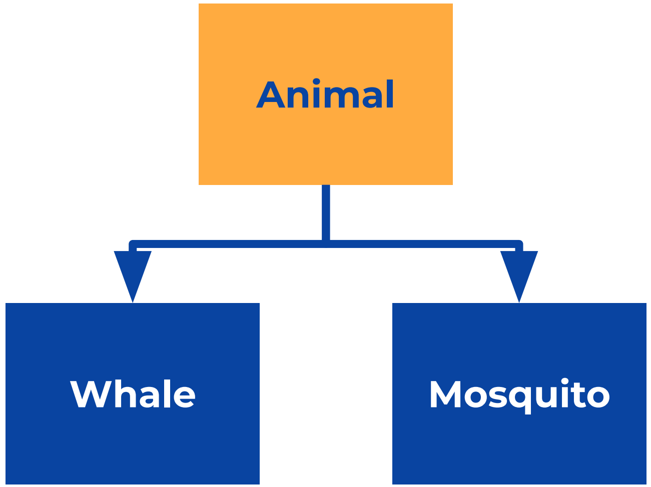
- It is not practical to use a class `Animal` to instantiate whales and mosquitoes.
- As animals, they both share common traits (life expectancy, ingestion, excretion,...).
- But defining a whale using the `Animal` class requires having to manually indicate all common traits of whales every time.



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

Digital Career Institute

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.

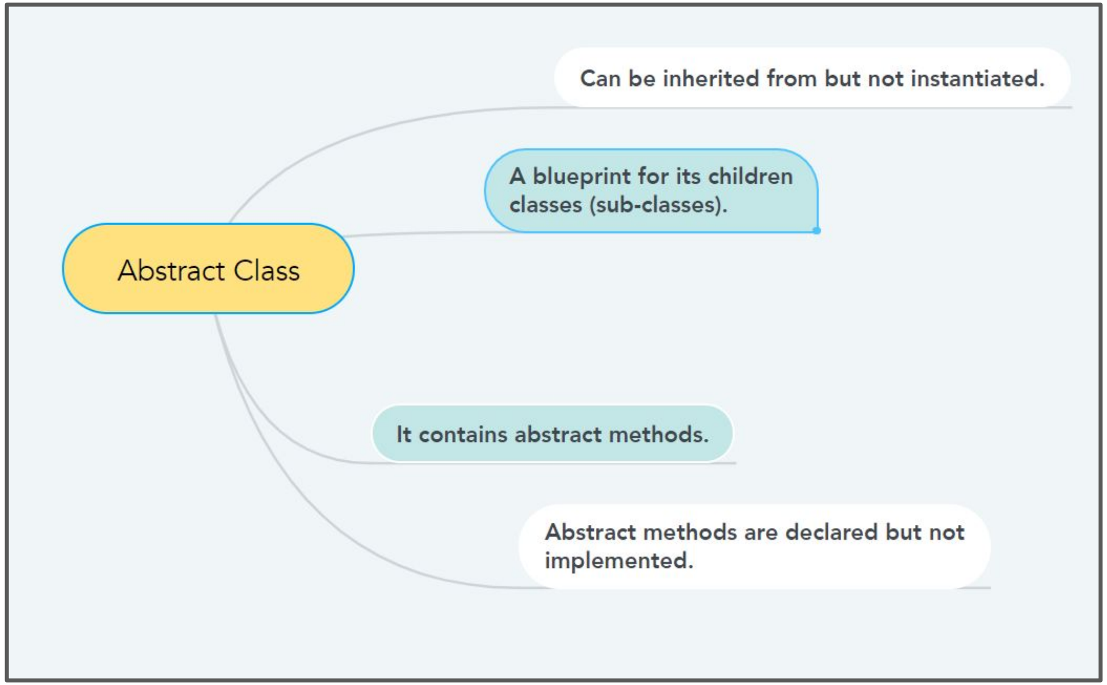
```
>>> from abc import ABC, abstractmethod
>>> class Animal(ABC):
...     @property
...     @abstractmethod
...     def reproduction(self):
...         pass
...
>>> class Whale(Animal):
...     pass
...
>>> willy = Whale()
TypeError: Can't instantiate abstract
class Whale with abstract method
reproduction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

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.

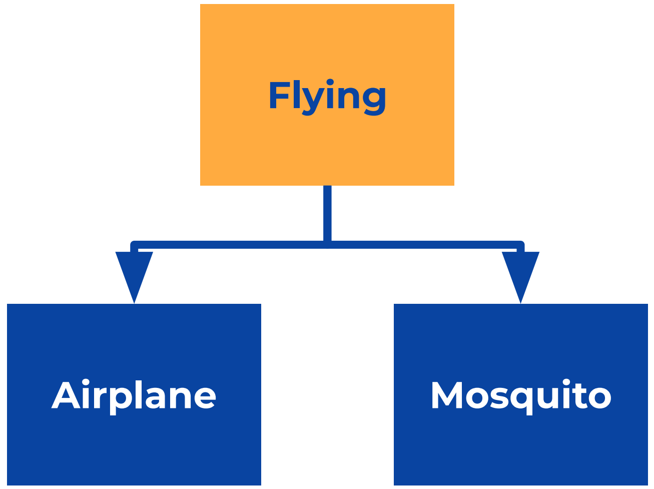
Mixins

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.

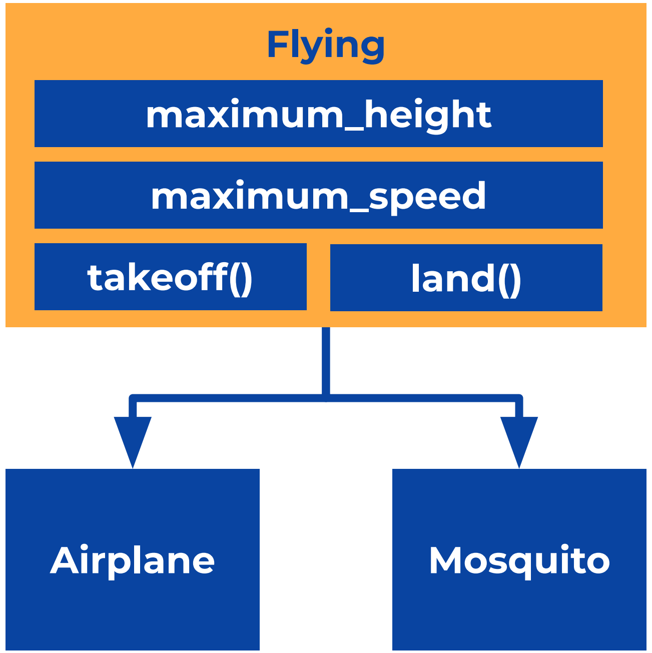


Class Mixins

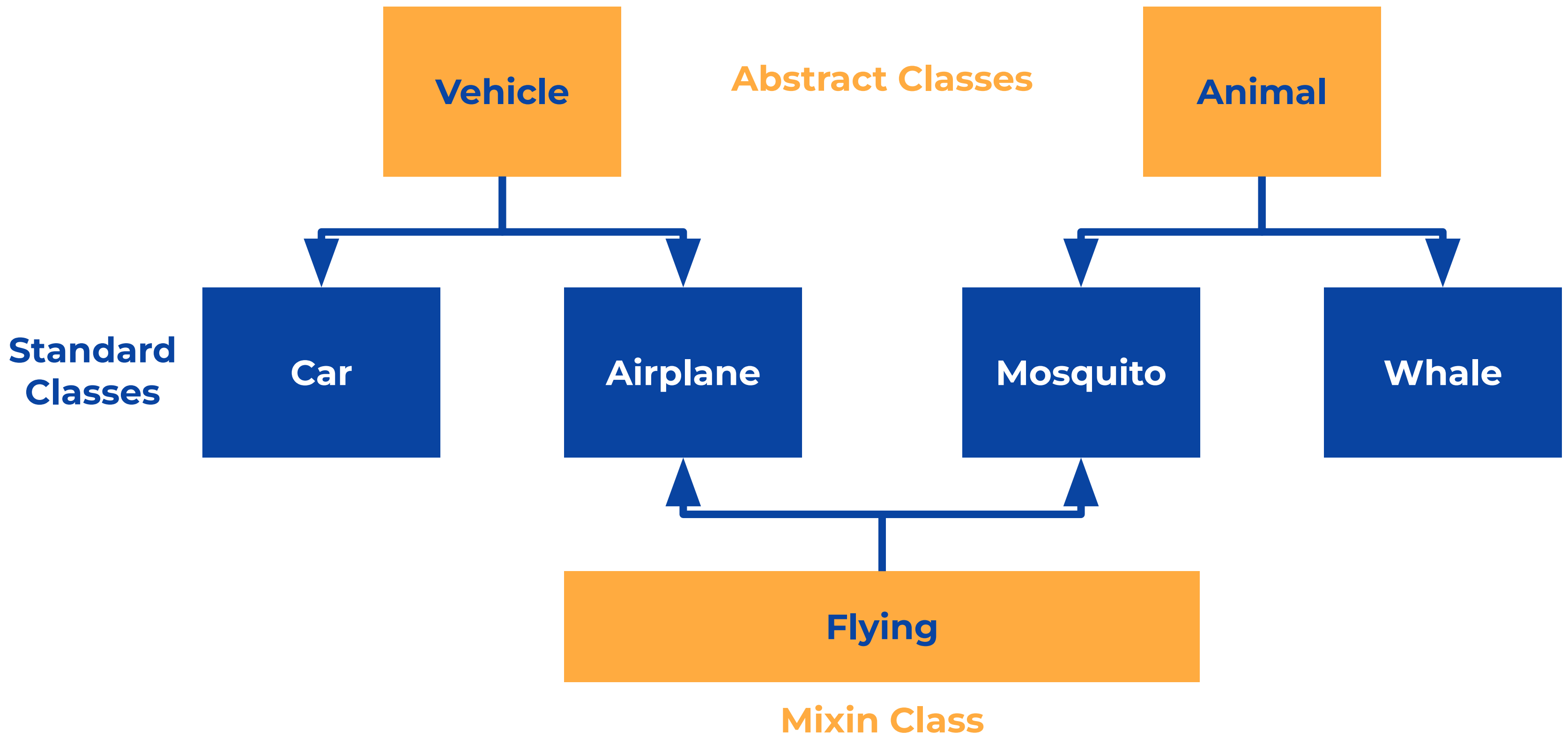
Mixins define and implement a **single, well-defined feature**.

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

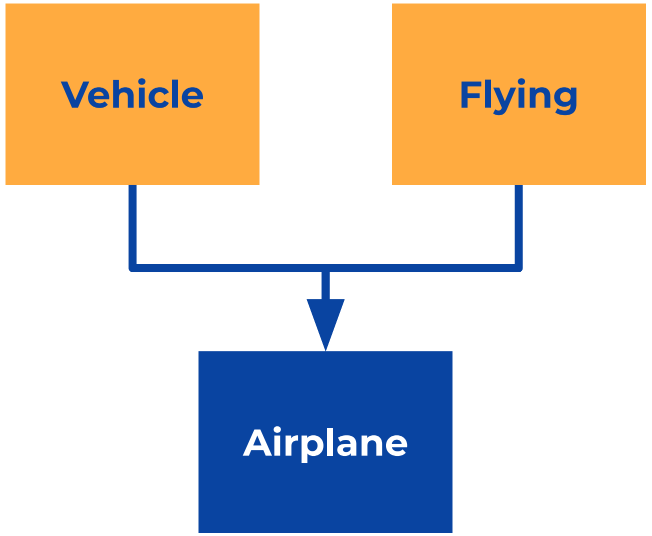


Python Mixins: Multiple Inheritance

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

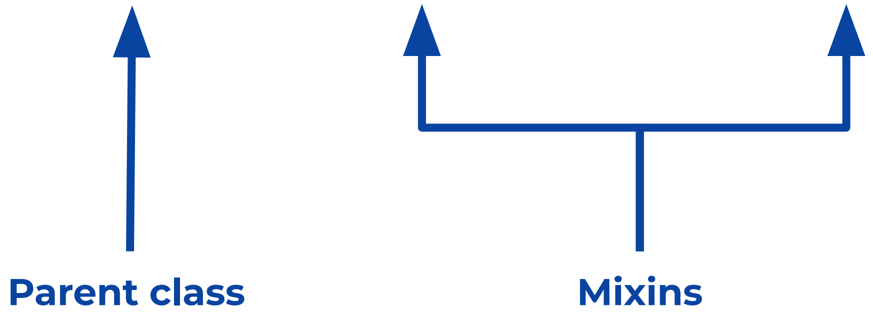
Multiple mixins can be used to extend a class.

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.

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute