LEARN PYTHON PROGRAMMING

How to Write Cleaner Python Code Using Abstract Classes

Petar Marković
5 min read · Feb 18

What are Abstract Classes? Why are they useful? When should you use them? Let me give you a few examples and explanations! By the end of this post, you'll have a firm understanding of ABCs in Python, and how to add them to your programs.

Let's begin!

Our code without abstract classes

I believe the best way to learn is with an example, so let's take a look at the following code:

```
class Lion:
    def give_food(self):
        print("Feeding a lion with raw meat!")

class Panda:
    def feed_animal(self):
        print("Feeding a panda with some tasty bamboo!")
```

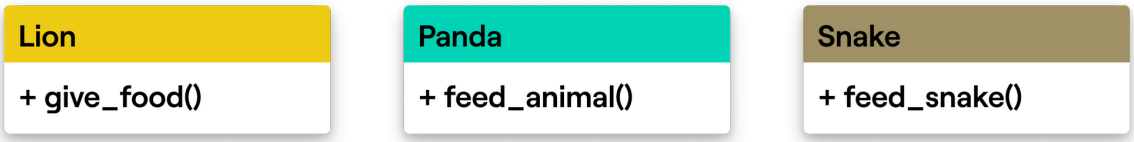
```
        print("Feeding a snake with mice!")

# Animals of our zoo:
leo = Lion()
po = Panda()
sam = Snake()
```

Our job is to feed all the animals using a Python script. One way to do that would be:

```
leo.give_food()
po.feed_animal()
sam.feed_snake()
```

This would work. But imagine how much time would it take to do this for each animal in a large zoo, repeating the same process and code hundreds of times. That would also make the code harder to maintain.

Currently our program's structure looks something like this:



We want to optimize the process, so we could come up with a solution like this one:

```
# Put all the animals in a list:
zoo = [leo, po, sam] # Could be many more animals there!
```

```
for animal in zoo:
    # But what do we put here now?
    # Is it animal.give_food() or animal.feed_animal(), hmm?
    animal.feed() # This will throw an AttributeError!
```

The problem is that every class has a different method name, when feeding a lion it's `give_food()`, when feeding a panda it's `feed_animal()` and it's `feed_snake()` for a snake.

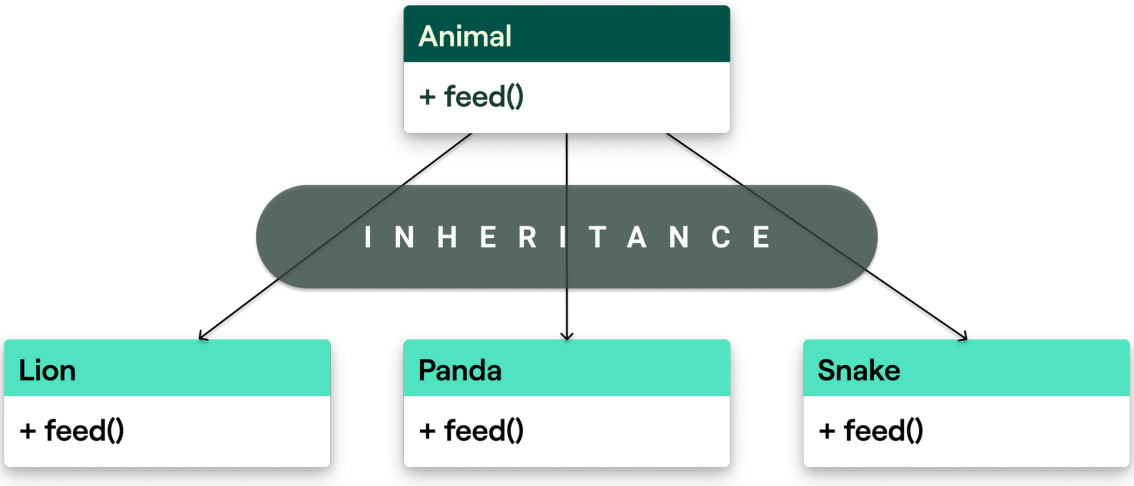
This code is a mess because methods that do the same thing should be named the same.

If we could only force our classes to implement the same method names...

Introducing Abstract classes

It turns out that the *Abstract class* is what we need. Essentially it forces its subclasses to implement all of its abstract methods. It is a class that represents what its subclasses look like.

A better structure could look like like this (`Animal` is an Abstract class):



from `Animal` **must** implement abstract methods from `Animal`, which in our case is the method `feed()`

Let's take a look at the code:

```
from abc import ABC, abstractmethod
# abc is a builtin module, we have to import ABC and abstractmethod

class Animal(ABC): # Inherit from ABC(Abstract base class)
    @abstractmethod  # Decorator to define an abstract method
    def feed(self):
        pass
```

When defining an abstract class we need to inherit from the Abstract Base Class - `ABC`.

To define an abstract method in the abstract class, we have to use a decorator: `@abstractmethod`. The built-in `abc` module contains both of these.

If you inherit from the Animal class but don't implement the abstract methods, you'll get an error:

```
class Panda(Animal): # If a class inherits from an ABC, it must implement all it's
    def wrong_name(self): # The method's name must match the name of the ABC's met
        print("Feeding a panda with some tasty bamboo!")
```

If we try to instantiate the class (e.g. `po = Panda()`) it will throw a `TypeError` since we can't instantiate Panda without an abstract method `feed()`.

correctly):

```
class Lion(Animal):
    def feed(self):
        print("Feeding a lion with raw meat!")

class Panda(Animal):
    def feed(self):
        print("Feeding a panda with some tasty bamboo!")

class Snake(Animal):
    def feed(self):
        print("Feeding a snake with mice!")
```

And lastly, this is all the code we need in order to create and feed our animals:

```
zoo = [Lion(), Panda(), Snake()]

for animal in zoo:
    animal.feed() # Now this won't throw an error!
```

Writing abstract methods with parameters

What happens when an abstract method has parameters? When the subclass implements the method, it must contain all the parameters as well. The subclass' implementation can also add extra parameters if required.

```
class Animal(ABC):
    @abstractmethod
    def do(self, action): # Renamed it to "do", and it has "action" parameter
        pass

class Lion(Animal):
    def do(self, action, time): # It's still mandatory to implement action. "time"
        print(f"{action} a lion! At {time}")

class Panda(Animal):
    def do(self, action, time):
        print(f"{action} a panda! At {time}")

class Snake(Animal):
    def do(self, action, time):
        print(f"{action} a snake! At {time}")
```

```
zoo = [Lion(), Panda(), Snake()]

for animal in zoo:
    animal.do(action="feeding", time="10:10 PM")
```

Running the above code will print out:

```
feeding a lion! At 10:10 PM
feeding a panda! At 10:10 PM
feeding a snake! At 10:10 PM
```

We could also use default arguments, you can read about those here.

We may also want to create abstract properties and force our subclass to implement those properties. This could be done by using `@property` decorator along with `@absctractmethod`.

Since animals often have different diets, we'll need to define a `diet` in our animal classes. Since all the animals are inheriting from `Animal`, we can define `diet` to be an abstract property. Besides `diet`, we'll make `food_eaten` property and it's `setter` will check if we are trying to feed the animal something that's not on it's `diet`.

Take a look at the code of `Animal`, `Lion` and `Snake`:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @property
    def food_eaten(self):
        return self._food

    @food_eaten.setter
    def food_eaten(self, food):
        if food in self.diet:
            self._food = food
        else:
            raise ValueError(f"You can't feed this animal with {food}.")

    @property
    @abstractmethod
    def diet(self):
        pass

    @abstractmethod
    def feed(self, time):
        pass

class Lion(Animal):
    @property
    def diet(self):
```

```
    def feed(self, time):
        print(f"Feeding a lion with {self._food} meat! At {time}")

class Snake(Animal):
    @property
    def diet(self):
        return ["frog", "rabbit"]

    def feed(self, time):
        print(f"Feeding a snake with {self._food} meat! At {time}")
```

We can create two objects, set the food that we're going to feed them and then call the `feed()` method:

```
leo = Lion()
leo.food_eaten = "antelope"
leo.feed("10:10 AM")
adam = Snake()
adam.food_eaten = "frog"
adam.feed("10:20 AM")
```

That will print out:

```
Feeding a lion with antelope meat! At 10:10 AM
Feeding a snake with frog meat! At 10:10 AM
```

If we try to feed an animal something that it doesn't eat:

```
leo.food_eaten = "carrot"
leo.feed("10:10 AM")
```

The `setter` will raise a `ValueError`:

```
You can't feed this animal with carrot.
```

A word on abstract Meta class

You may have come across metaclasses when learning about abstract classes.

A class defines how an instance of the class behaves (e.g. `Animal` describes how `Lion` will behave). On the other hand a metaclass defines how a class behaves (`ABCMeta` describes how every `ABC` class will behave). A class is an instance of a metaclass.

The `abc` module comes with a metaclass `ABCMeta`. back in the days we had to use it to define metaclasses with `metaclass=abc.ABCMeta`. Nowadays, just inheriting from `ABC` does the same thing—so you don't have to worry about metaclasses at all!

Summary

In this blog post we described the basics of Python's abstract classes. They are especially useful when working in a team with other developers and parts of the project are developed in parallel.

Here are some key takeaways:

methods and properties defined in the abstract base class.

- Abstract base class can't be instantiated.
- We use `@abstractmethod` to define a method in the abstract base class and combination of `@property` and `@abstractmethod` in order to define an abstract property.

I hope you learnt something new today! If you're looking to upgrade your Python skills even further, check out our Complete Python Course.

Member discussion

0 comments

Start the conversation

Become a member of **The Teclado Blog** to start commenting.

Sign up now

Already a member? **Sign in**

Petar Marković

Hello! I'm a teaching assistant and a content writer at Teclado. I'm at the beginning of my career and I am currently learning and working with Python.

Read More

Learn Python Programming

How to use pyenv to manage Python versions →

How to run tasks periodically in Render.com →

How to upload files to Backblaze B2 using Python →

→ See all 154 posts

Python Methods: Instance, Static and Class

What are @classmethod and @staticmethod? How are they different from instance methods? Learn about the three types of Python method in this post!

Vlad Dragusanu
6 min read · Mar 3