Theory: Abstract classes

Suppose, you want to create a role-playing game. You have come up with a bunch of different character classes and you want to define their actions. You want your characters to explore the world, interact with each other, fight, cast spells, sing songs. All characters should be able to do all these things, but the exact way they do it should depend on the type of character.

In practice, this means that you need to create a class for each character and define the corresponding methods. To make the process easier and more structured, you should use **abstract classes.**

In this topic, we will discuss what abstract classes are, and why they are perfect for these tasks.

§1. What are abstract classes?

Generally speaking, an **abstract class** is a template that can be used to create other classes. Once we have a template, we do not work with it directly, we create other objects based on this template and work with them instead. Abstract classes operate in a similar manner.

So, what makes a class abstract?

Well, for one, we cannot create instances of abstract classes. Since an abstract class is some kind of a blueprint, it would make no sense to create such an instance. Another feature of abstract classes is that they have **abstract methods.** Abstract methods are methods that generally do not have any implementation and they are declared with the `@abstractmethod` decorator.

You may wonder what is the purpose of these abstract classes since there are no objects and no functionality. Well, their value lies in the fact that they define the structure and functionality for other classes. Abstract classes are used as parent or base classes. All abstract methods defined in the abstract class should be overridden in a child class.

> Note that even though abstract methods generally do not have any implementation, they can be implemented. You would still need to override this method in the child class, though. So, implementing an abstract method doesn't make a lot of sense.

Take our RPG as an example. We can use abstract classes to create our characters: we would need to create an abstract player class, list all possible actions as methods, and then create child classes for specific character roles.

§2. How to create an abstract class?

To create an abstract class in Python, we need to use the `abc` module (which we first have to import). `abc` is a module for **a**bstract **b**ase **c**lasses, that is why we have the name.

The first step to make a class abstract is to declare it with a parent class `ABC` from the `abc` module.

```
1  from abc import ABC, abstractmethod
2
3
4  class Player(ABC):
5      ...
```

As we start to create our player classes, we take the class `Player` as our template. However, this is not enough. This class is not abstract yet as it does not have any abstract methods. So, you could create an instance of this class but it is not what we want.

Now we need to define methods, they represent the actions that our players will be able to do.

```
from abc import ABC, abstractmethod


class Player(ABC):
    def __init__(self, name, rank, level):
        self.name = name
        self.rank = rank
        self.level = level
        super().__init__()



    @abstractmethod

    def fight(self):

        ...



    @abstractmethod

    def do_spell(self):

        ...


    def sing_song(self):

        print("No songs for me!")
```

Now, this is a proper abstract class — it inherits from `ABC` and has abstract methods.

> Note that not all methods in the abstract class need to be abstract. With the abstract class, only "mandatory" methods should be abstract because they will need to be overridden in the child class to work properly.

If we attempt to create an object of this class now, we will get a `TypeError`:

```
some_player = Player("Legolas", 3, 3)


# TypeError: Can't instantiate abstract class Player with abstract methods do_spel
l, fight
```

§3. Subclasses

Now we have a proper template, which means we are ready to create actual player classes. Let's start with the class `Warrior`.

```
class Warrior(Player):
    ...
```

We have already mentioned that abstract methods need to be overridden in the subclasses of an abstract class. What happens if they remain the way they are? Well, as you can see, with the class `Warrior` above, we have not overridden anything. This is what happens when we try to create an object of this class:

```
warrior = Warrior("Bran", 1, 1)


# TypeError: Can't instantiate abstract class Warrior with abstract methods do_spe
ll, fight
```

We have an error because we have not overridden abstract methods `do_spell` and `fight`. Note that nothing is said about the `sing_song` method since it is not abstract.

Now, let's write this class properly:

```
class Warrior(Player):
    def fight(self):
        print("Swing an ax")

    def do_spell(self):
        print("Increase weapon fatality")


warrior = Warrior("Bran", 1, 1)



warrior.fight()

# Swing an ax

warrior.do_spell()

# Increase weapon fatality

warrior.sing_song()

# No song for me!
```

No exceptions now! Here, you can see that the `sing_song` method was inherited "as is" from the `Player` class. Since it is a regular instance method, we do not have to override it in the child class. But we could do it just in case.

Take another character class, `Bard`, as an example. Bards do need to sing, so we could override this method:

```
class Bard(Player):
    def fight(self):
        print("Smash the opponent with your lute.")

    def do_spell(self):
        print("Enchant everyone with your tale.")

    def sing_song(self):
        print("Sing a beautiful song.")







bard = Bard("Jaskier", 4, 5)



bard.fight()

# Smash the opponent with your lute.

bard.do_spell()

# Enchant everyone with your tale.

bard.sing_song()

# Sing a beautiful song.
```

Now our bards can sing a song!

§4. Summary

To sum up:

- Abstract classes serve as a template for other classes.

- Abstract classes inherit from the class `ABC` from `abc` module and have abstract methods.
- Abstract methods have no implementation. They are preceded by `@abstractmethod` decorator.
- Abstract methods should be overridden in child classes.

Report a typo

Start practicing

Comments (5) Hints (0) Useful links (0) Show discussion