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!")

class Snake:

def feed\_snake(self):

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:

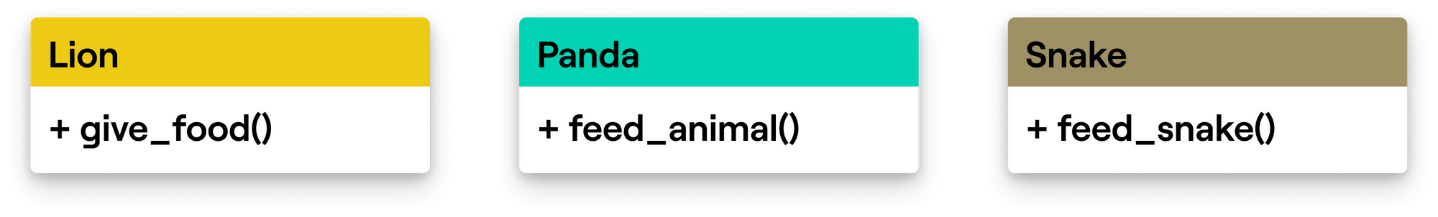
leo.give\_food()

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!

# Loop through the animals and feed them

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.

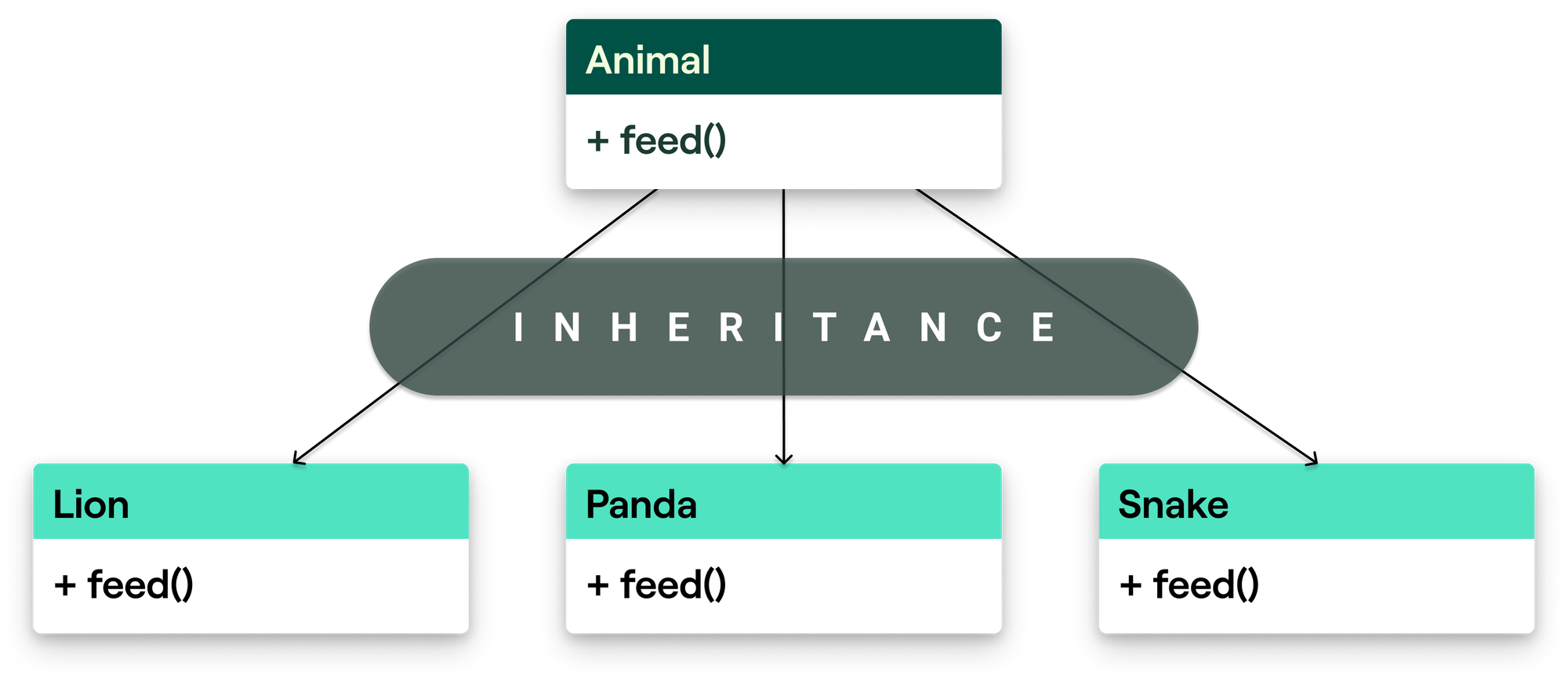
This code is a mess because methods that do the same thing should be named the same.

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):



By introducing an abstract class (Animal), every class that inherits 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 abstract methods!

def wrong\_name(self): # The method's name must match the name of the ABC's method

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().

Keeping that in mind, we need to make our animals (this time 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.

from abc import ABC,abstractmethod

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" is our other parameter

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](https://python.tecladocode.com/4_treasure_hunters/5_default_parameter_values.html#the-order-of-default-parameters).

**Writing (abstract) properties**

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):

return ["antelope", "cheetah", "buffaloe"]

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 = Lion()

leo.food\_eaten = "carrot"

leo.feed("10:10 AM")

The setter will raise a ValueError:

You can't feed this animal with carrot.