Python Intermediate

OOP - Methods, Getters and Setters

{codenation}®

Extending with Sub Classes

Base classes

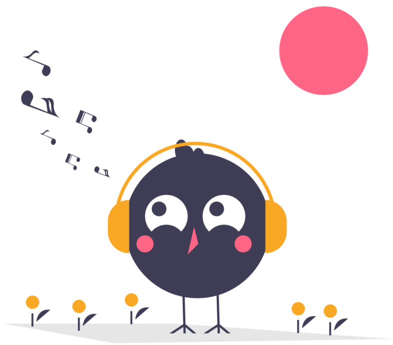
In some cases, one class might not be able to provide enough structure to be useful.

Try to create a blueprint for a pet.
What properties and methods do pets have?

Base classes

There are too many different types of pets!

Creating a class that covers all of them would be tough.

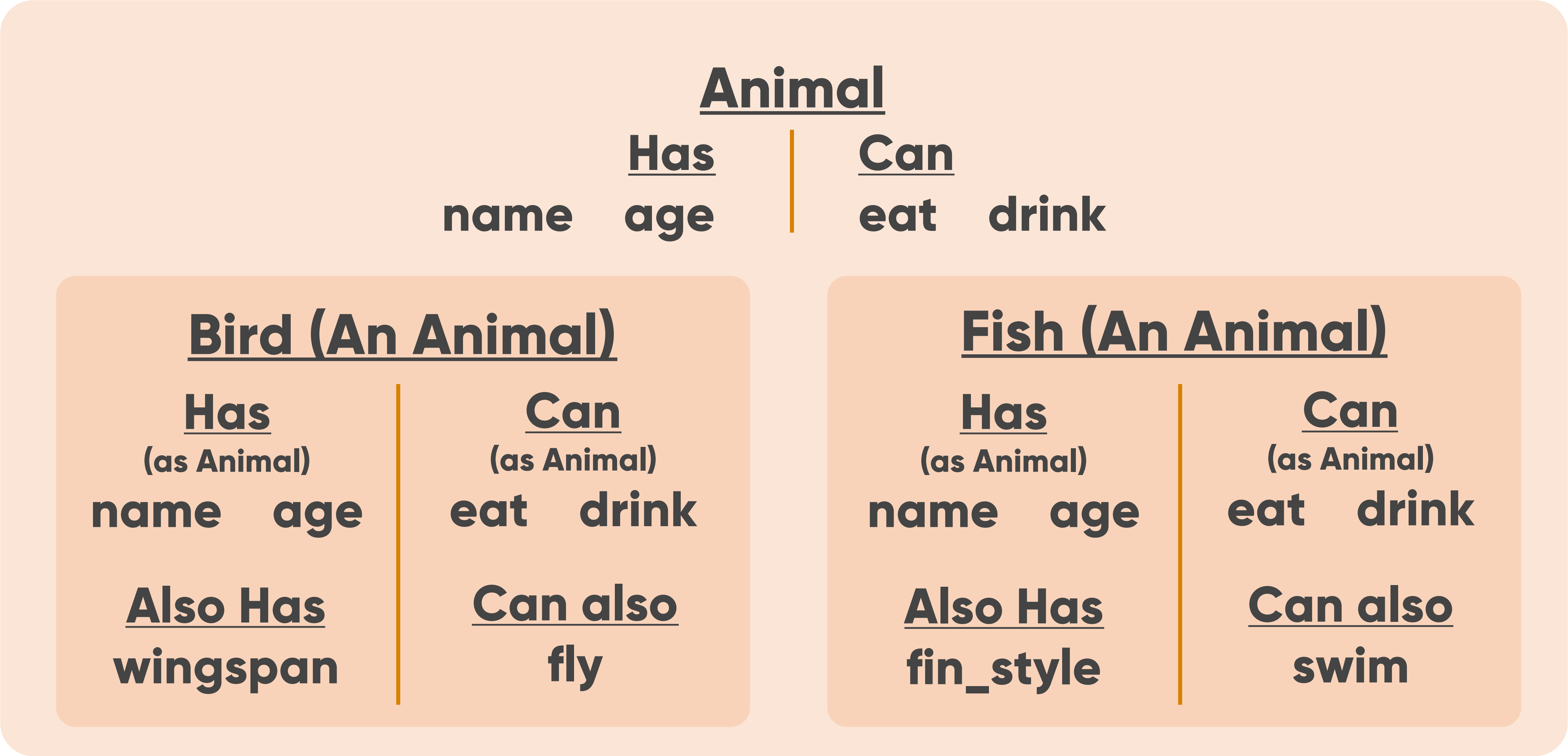


Base classes

OOP focuses on **inheritance**.

We can build parent classes and subclasses.

Subclasses will **inherit** all the properties and methods from the parent class they are based on but allow us to safely add more specific ones.



Animal

Has
name age

Can
eat drink

Bird (An Animal)

Has
(as Animal)
name age

Also Has
wingspan

Can
(as Animal)
eat drink

Can also
fly

Fish (An Animal)

Has
(as Animal)
name age

Also Has
fin_style

Can
(as Animal)
eat drink

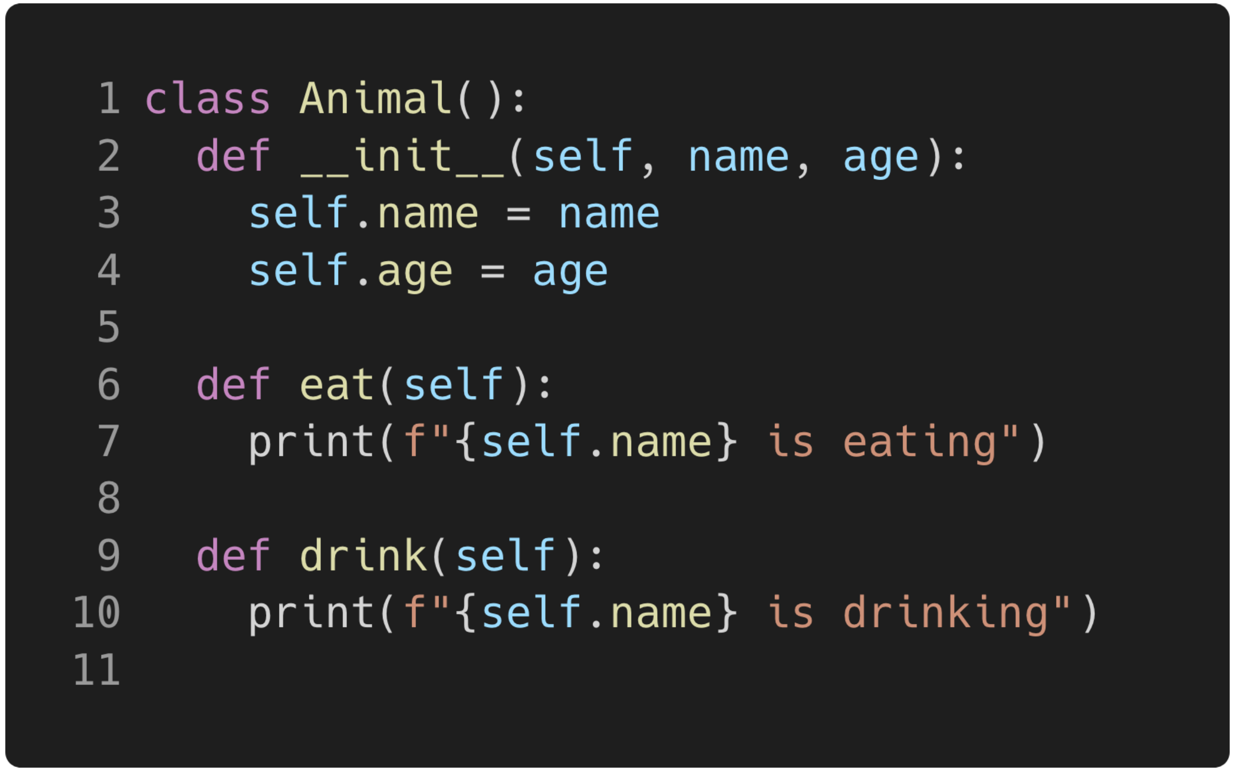
Can also
swim

Inheritance

We can safely say all pets are **Animals**.

All our pets will have a name and an age.

All our pets will have the ability to eat and drink.

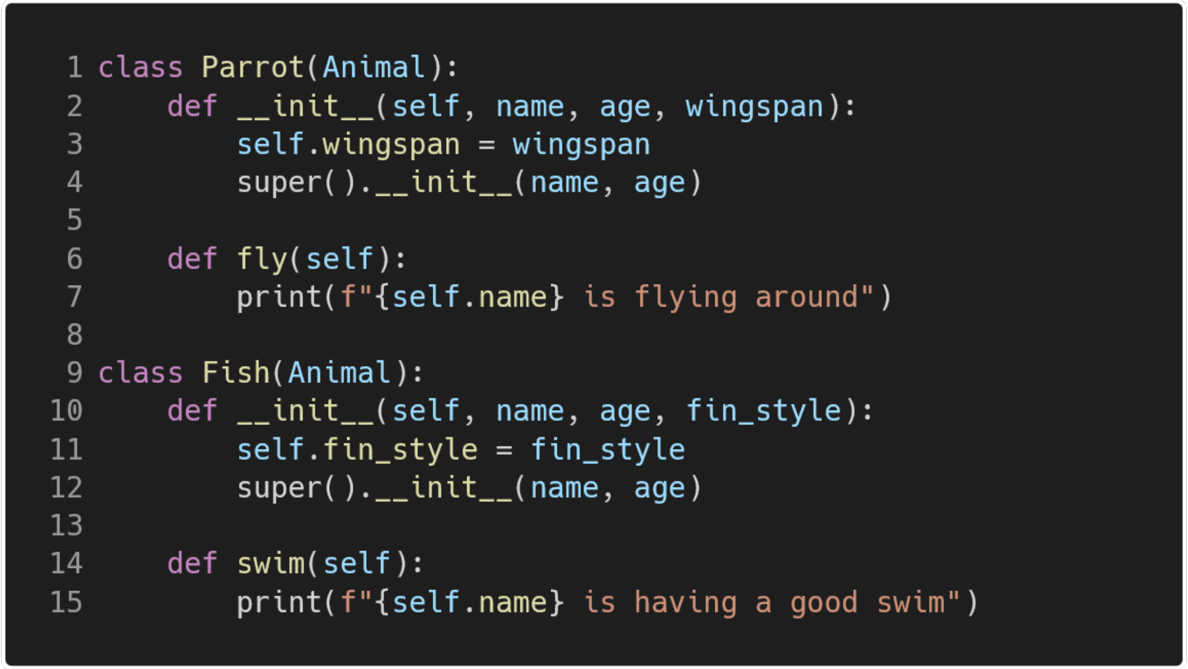


```python
class Animal():
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def eat(self):
    print(f"{self.name} is eating")

  def drink(self):
    print(f"{self.name} is drinking")

```

Inheritance

We can make two more specific classes – **Parrot** and **Fish**.

Both inherit from **Animal**, so will have names, ages, and the ability to eat and drink.



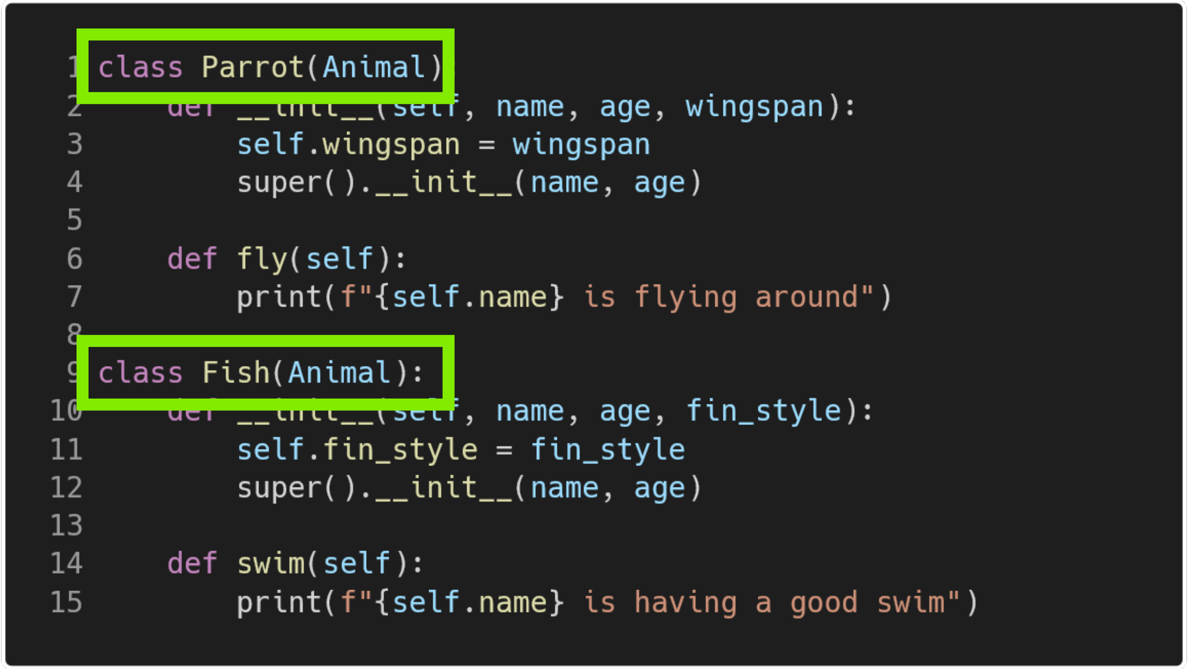
```python
class Parrot(Animal):
    def __init__(self, name, age, wingspan):
        self.wingspan = wingspan
        super().__init__(name, age)

    def fly(self):
        print(f"{self.name} is flying around")

class Fish(Animal):
    def __init__(self, name, age, fin_style):
        self.fin_style = fin_style
        super().__init__(name, age)

    def swim(self):
        print(f"{self.name} is having a good swim")
```

Inheritance

These are sub classes.

The **extend** the functionality of the parent or base class **Animal**.

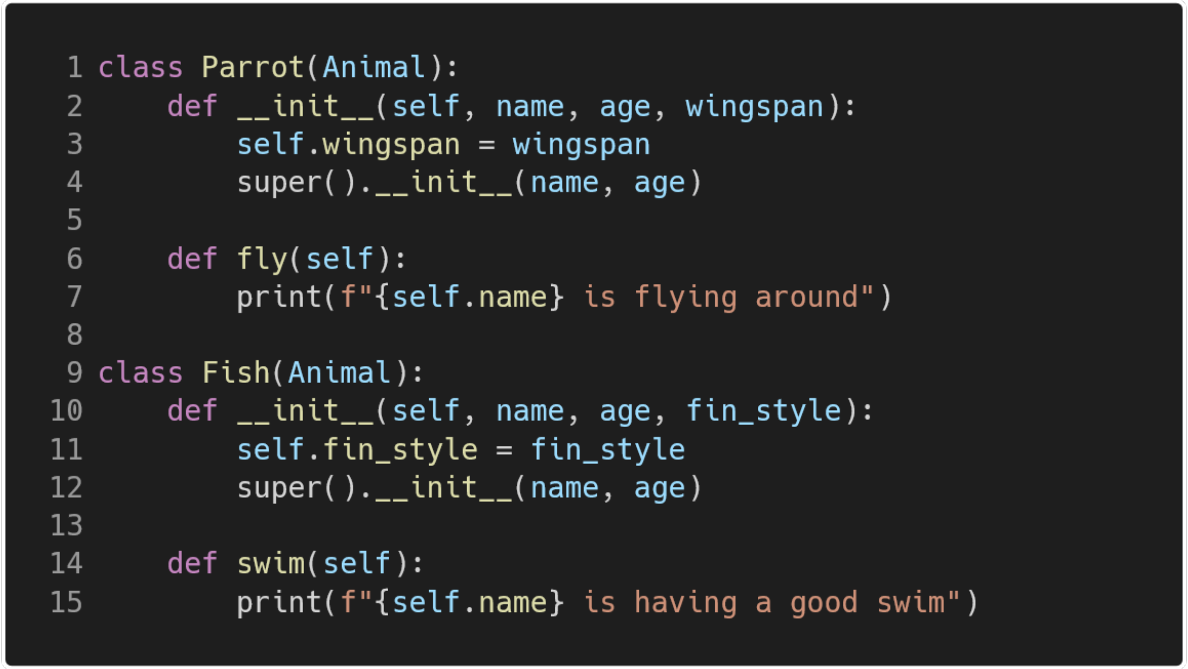
They should always be used to improve **Animal**, not to take things away.



```python
class Parrot(Animal):
    def __init__(self, name, age, wingspan):
        self.wingspan = wingspan
        super().__init__(name, age)

    def fly(self):
        print(f"{self.name} is flying around")

class Fish(Animal):
    def __init__(self, name, age, fin_style):
        self.fin_style = fin_style
        super().__init__(name, age)

    def swim(self):
        print(f"{self.name} is having a good swim")
```

Inheritance

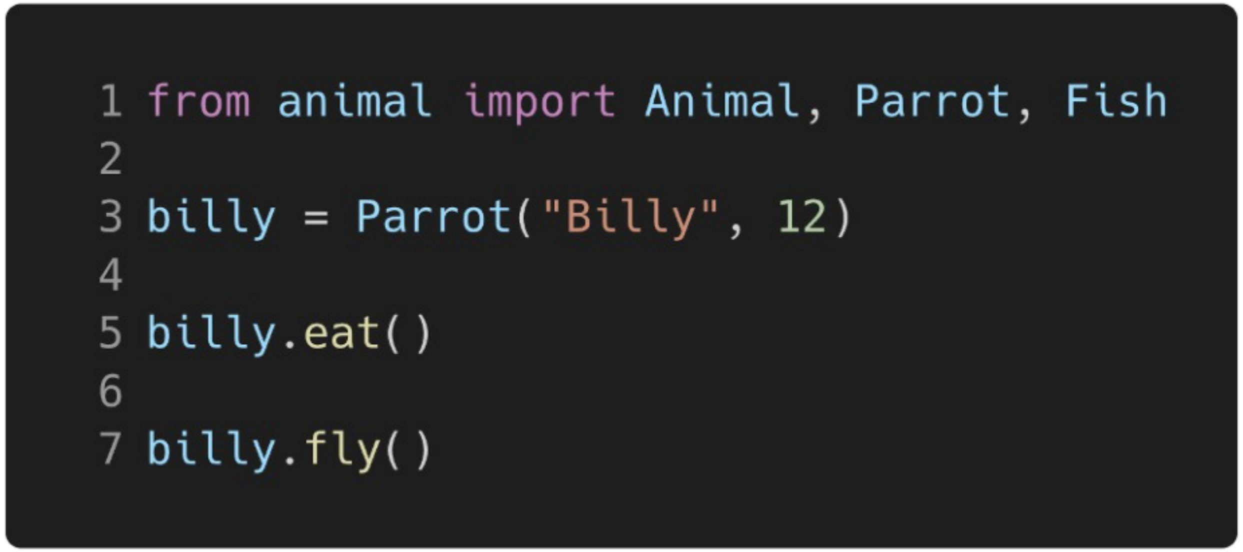
They also have their own unique properties and methods.



```python
class Parrot(Animal):
    def __init__(self, name, age, wingspan):
        self.wingspan = wingspan
        super().__init__(name, age)

    def fly(self):
        print(f"{self.name} is flying around")

class Fish(Animal):
    def __init__(self, name, age, fin_style):
        self.fin_style = fin_style
        super().__init__(name, age)

    def swim(self):
        print(f"{self.name} is having a good swim")
```

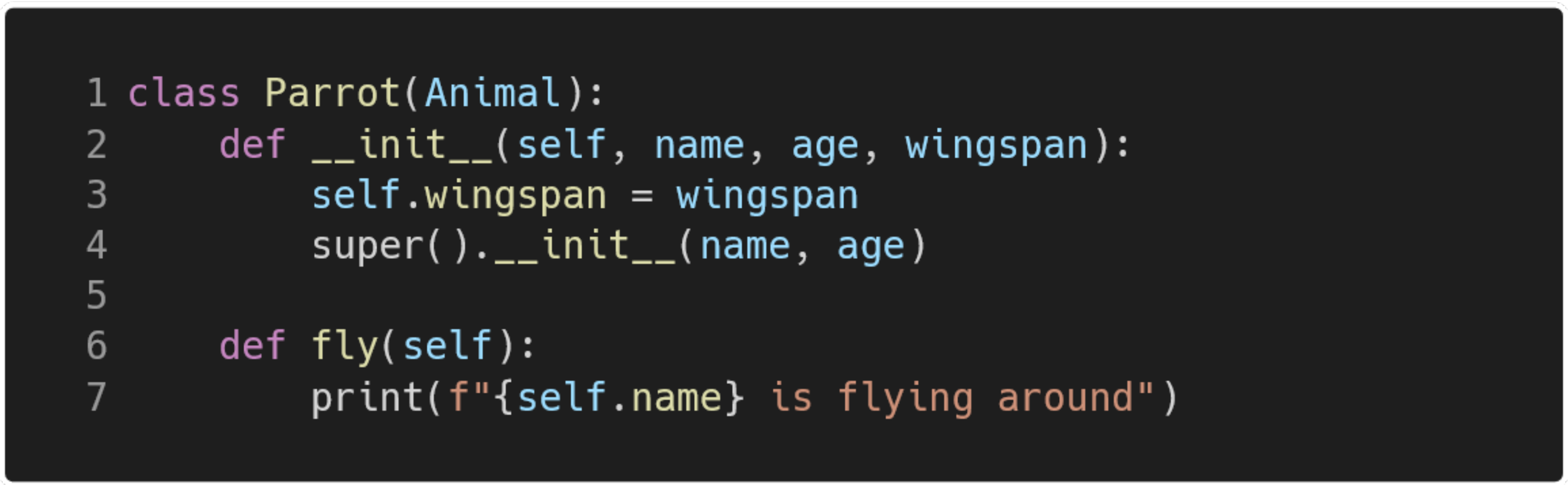
Inheritance

When we **instantiate** a **Parrot** object, we still need to give it a name and age, as well as the Parrot only property **wingspan**.



```python
from animal import Animal, Parrot, Fish

billy = Parrot("Billy", 12)

billy.eat()

billy.fly()
```

Inheritance



```python
class Parrot(Animal):
    def __init__(self, name, age, wingspan):
        self.wingspan = wingspan
        super().__init__(name, age)

    def fly(self):
        print(f"{self.name} is flying around")
```

The Parrot class’s __init__ function overwrites the Animal class’s __init__ function.

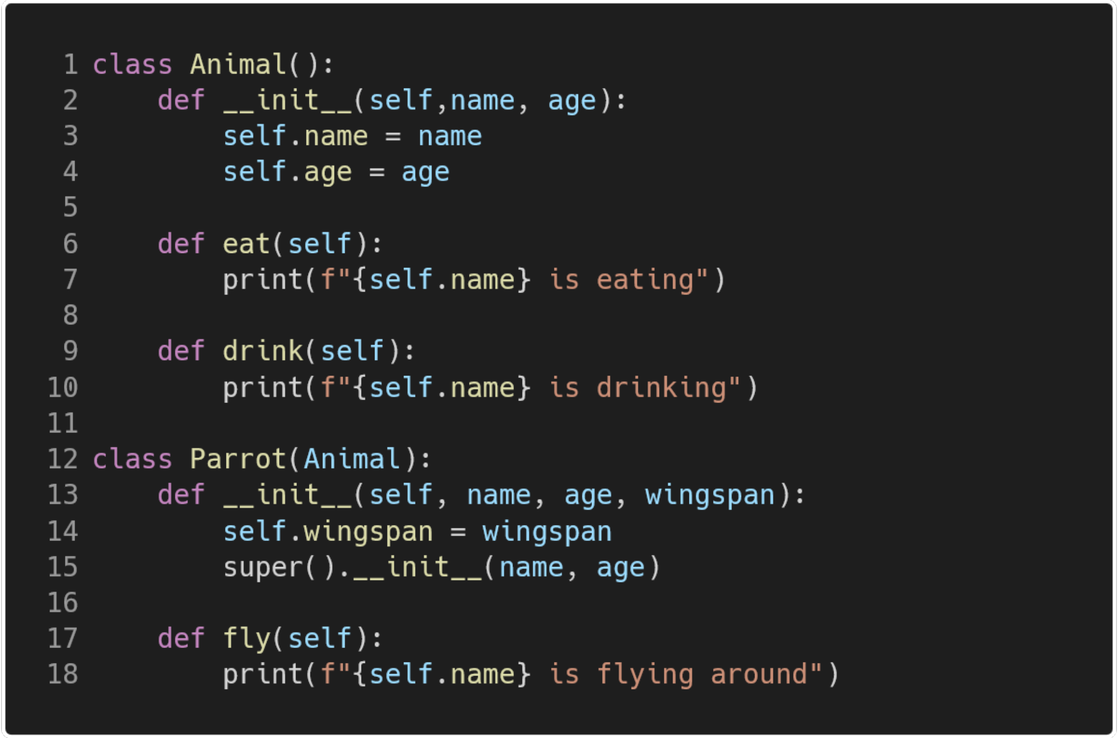
We recall the parent constructor using super().

Inheritance

In **Parrot** we only map out the **wingspan** property.

name and **age** are mapped out in **Animal**.

We pass them up to the parent on line 15.

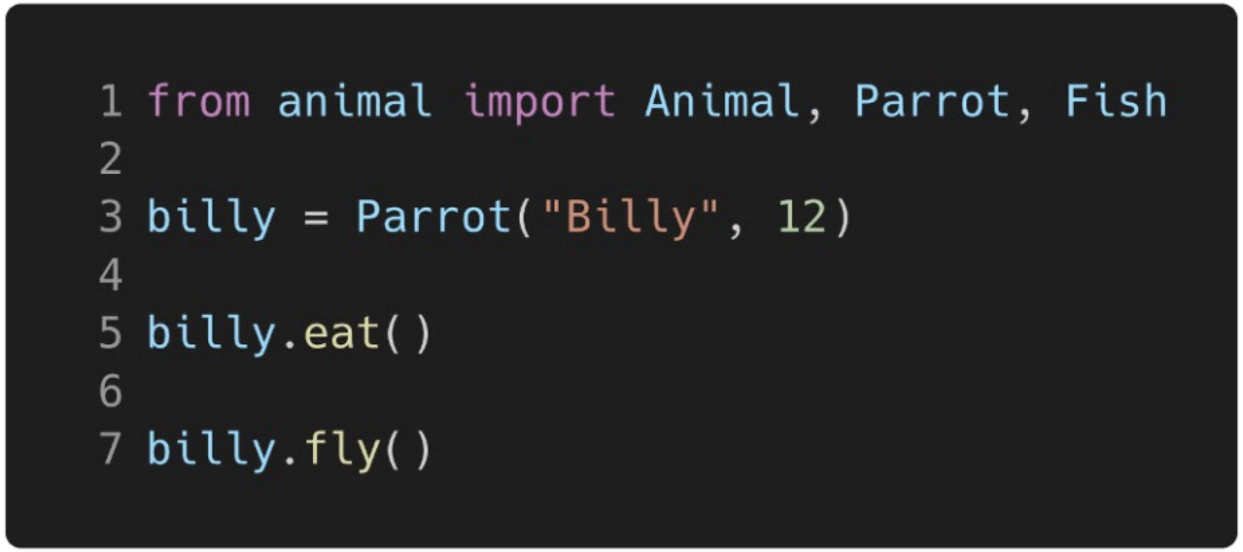


```
 1 class Animal():
 2     def __init__(self,name, age):
 3         self.name = name
 4         self.age = age
 5 
 6     def eat(self):
 7         print(f"{self.name} is eating")
 8 
 9     def drink(self):
10         print(f"{self.name} is drinking")
11 
12 class Parrot(Animal):
13     def __init__(self, name, age, wingspan):
14         self.wingspan = wingspan
15         super().__init__(name, age)
16 
17     def fly(self):
18         print(f"{self.name} is flying around")
```

Inheritance

Our **Parrot** can still eat. He has **inherited** these from the parent class of **Animal**.

But he can also fly.



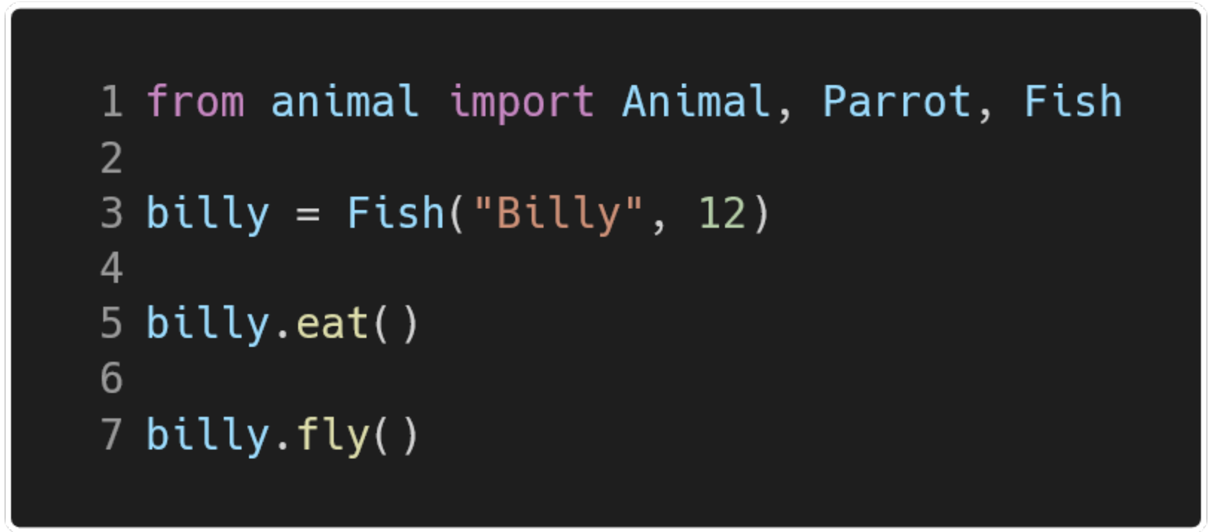
```python
from animal import Animal, Parrot, Fish

billy = Parrot("Billy", 12)

billy.eat()

billy.fly()
```

Inheritance

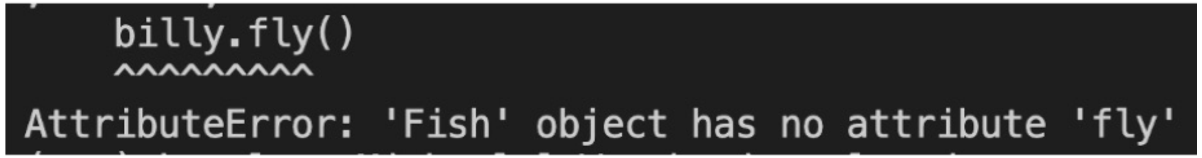
If **Billy** was a fish, he could not fly.

Based on our classes, only birds can fly, so **Billy** the **Fish** does not have access to the **.fly()** method.

The **I** in **SOLID** stands for the **Interface Segregation Principle**.



```python
from animal import Animal, Parrot, Fish

billy = Fish("Billy", 12)

billy.eat()

billy.fly()
```



```
    billy.fly()
    ^^^^^^^^^^
AttributeError: 'Fish' object has no attribute 'fly'
```

Inheritance

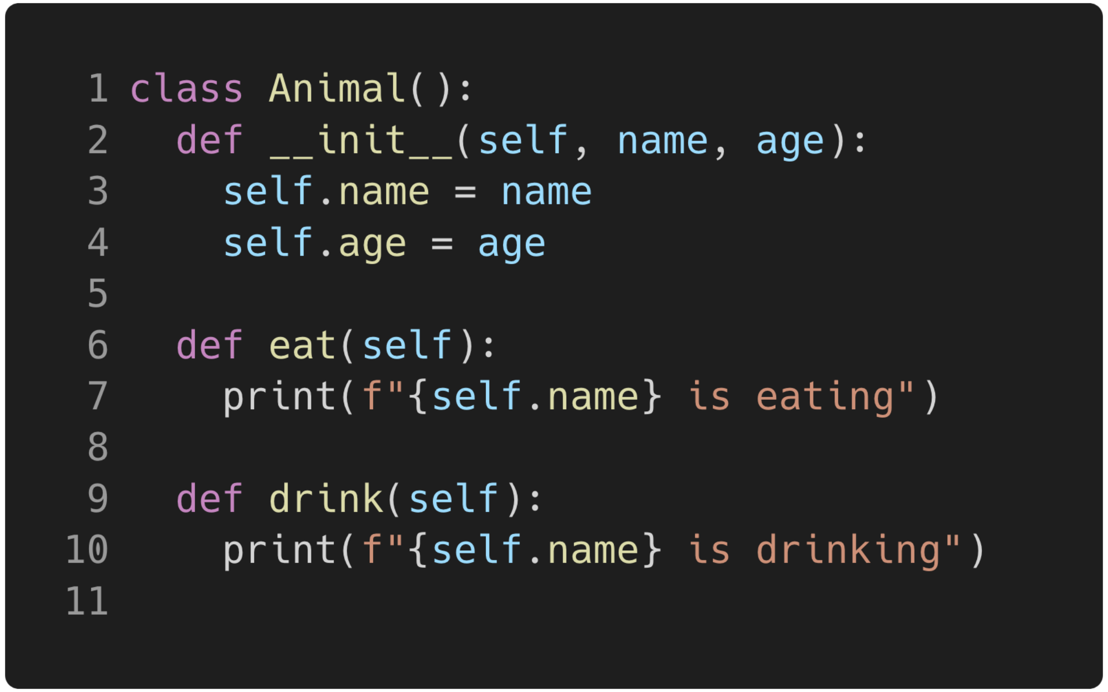
It would be wrong of us to code things related to flying into our base class, Animal.

Not every pet can fly.
If we put flying into our base class, objects might end up interacting with things they shouldn't.

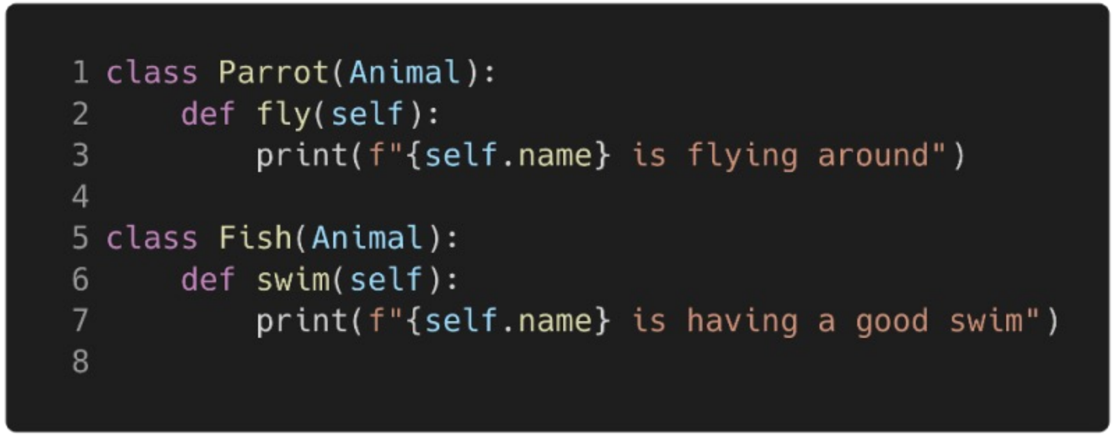
It is safer to keep flying-related processes accessible only to things that can fly.

Inheritance

By creating a parent class and allowing subclasses to inherit from it, we can write structured and reusable code without being limited!



```python
class Animal():
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def eat(self):
    print(f"{self.name} is eating")

  def drink(self):
    print(f"{self.name} is drinking")

```



```python
class Parrot(Animal):
    def fly(self):
        print(f"{self.name} is flying around")

class Fish(Animal):
    def swim(self):
        print(f"{self.name} is having a good swim")

```

Challenge 1

Create a class called **Vehicle**.
Define the properties and methods all vehicles have.

Pick two more specific types of vehicles (e.g. cars and aeroplanes).

Create **subclasses** for your chosen vehicles with more specific properties and methods.