INFO5002: Intro to Python for Info Sys

Week 5

Slides created by: Zachary Doucet

LVX
VERITAS
VIRTVS

Northeastern
University

Week 5

I. Advanced Data Types Practice

II. Introspection

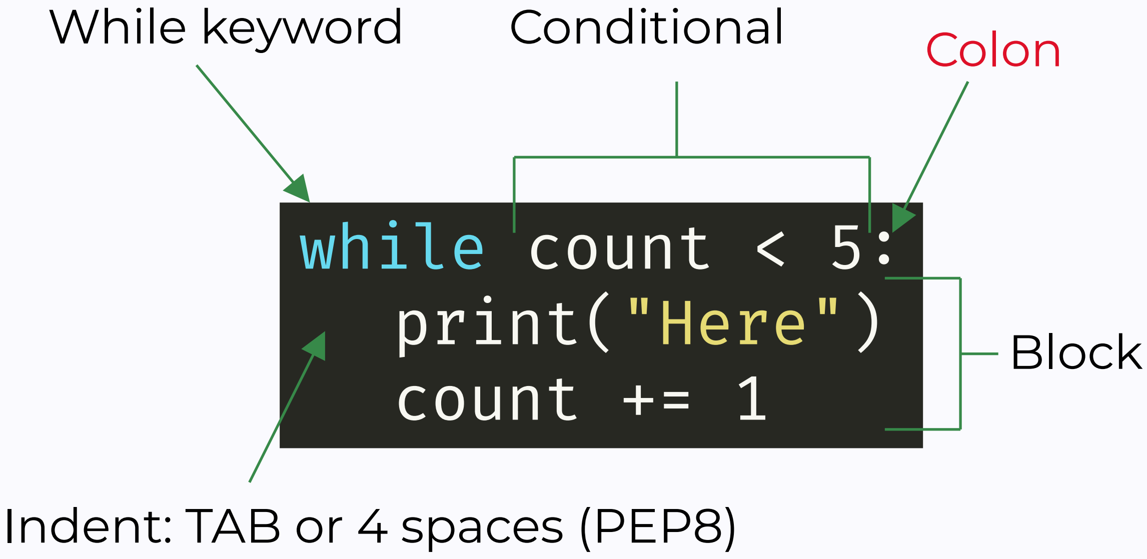
III. Classes

IV. OOP

Recap

Loops reduce repetition

- We repeat a group of operations together under a loop to reduce re-writing.



While keyword

Conditional

Colon

```
while count < 5:
    print("Here")
    count += 1
```

Block

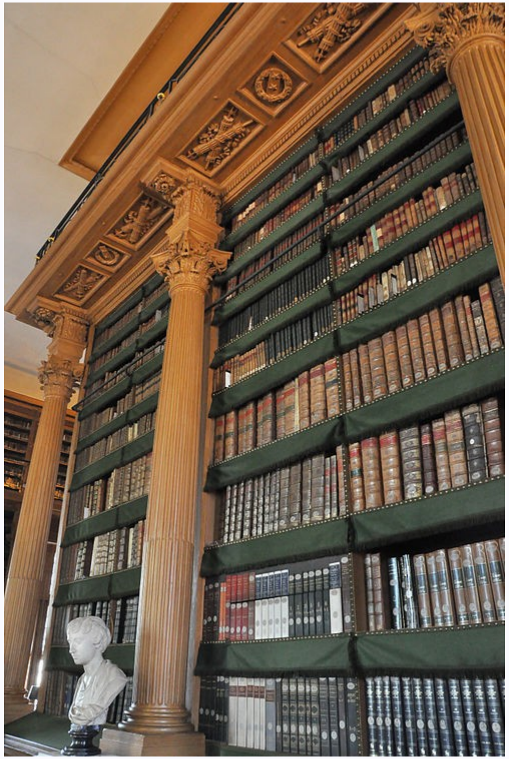
Indent: TAB or 4 spaces (PEP8)

Change loop path

- You can change the executing path with break and continue.
- Break exits the loop.
- Continue skips this iteration.

Collections

- Lists []
- Tuples ()
- Dictionaries {key: value}
- Sets {}



Source: Wikimedia

Advanced Data Types Practice

Let's practice

- Create the following function:

  I. sum which takes a list and returns the sum of all the values in the list.

  II. collect_stats which takes in a list of numbers and returns a tuple of the average and the median.

  III. remove_odd which takes a list and returns a new list without the odd numbers.

  IV. percent_passed which takes a list of exam grades and returns the percent of students that passed, assuming a pass >= 60.

And some more

- Create the following function:

  I. even_sum which takes a number n and returns the sum of all even numbers between 0 and n. You must use for loop.

  II. average_position which takes in an array of tuples and returns the average (x, y).

  III. remove_duplicate which takes in a list and returns a list with the duplicates removed.

Introspection

Inspect documentation

The ability to verify static type

- A variable can point to any type!
- Introspection allows you to collapse all possibles to a single type.

Source: Wikimedia

Check type w/ special functions

- type

```
x = "orange"
type(x)
```

```
x = 123
type(x)
```

```
x = 3.14
type(x)
```

```
if type(x) is str:
    print("I am string")
elif type(x) is int:
    print("I am int")
elif type(x) is float:
    print("I am float")
```

- callable

```
x = "banana"
callable(x)
```

```
def add(x, y):
    return x + y
x = add
callable(x)
```

Much more with inspect module

- inspect.isfuntion(object)
- inspect.signature(object)
- inspect.getsource(object)
  - Cannot pass in built-in functions

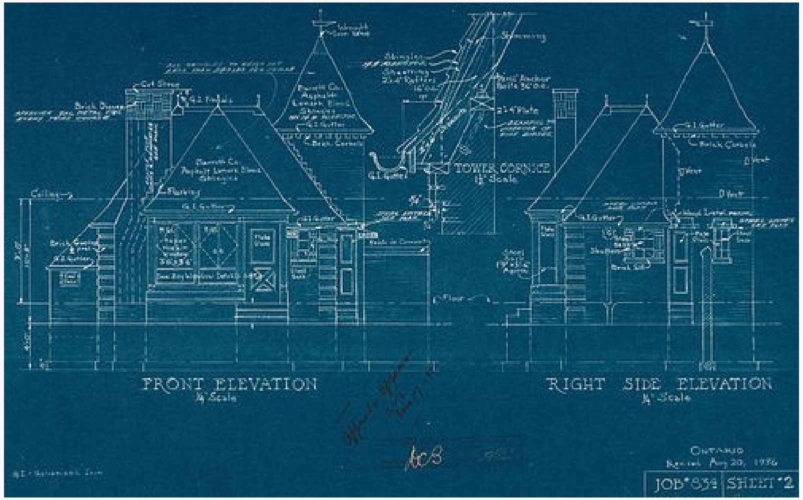
Classes

PCC 157-181

Classes to group functionality

- Functions group operations together and collections group data together.

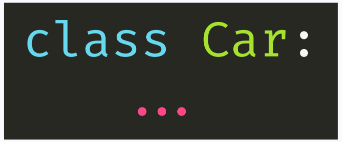
- Classes allow you to collect functions and variables together.



Source: City of Toronto Archives

Creating classes

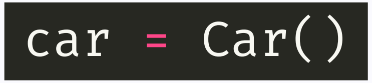
- Create a class with the class keyword.



```
class Car:
    ...
```

Creating objects

- Objects are created by “calling” a class.



```
car = Car()
```

Classes vs Objects

- Classes are like function declarations and objects are like function calls.
- Objects are instantiated classes.
- Objects have state which is the current value of all variables held by the object.

To avoid confusion, name properly!

- Variables and functions follow snake_case.

- For **classes** use UpperCamelCase.

```
class LightSaber:
    pass
```

```
class BuildingMaterial:
    pass
```

```
class PlanetaryVehicle:
    pass
```

Give your classes statefulness

- You can add attributes to your classes.

```
class myClass:
    x = 1
```

- You can turn your dictionaries (with finite keys) into classes.

```
x = {"model": "Kia Rio",
  "year": 2003, "mpg": 25.32}
```

```
class Car:
    model = "Kia Rio"
    year = 2003
    mpg = 25.32
x = Car()
```

Accessing and Mutating

```
x["model"]
```

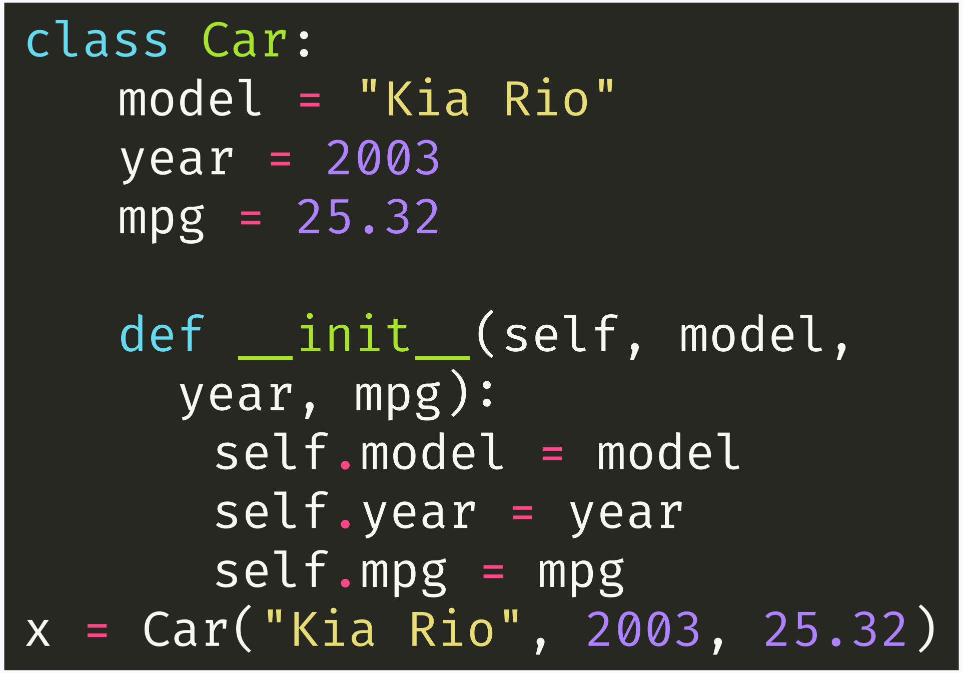
```
x.model
```

```
x["model"] = "Kia Soul"
```

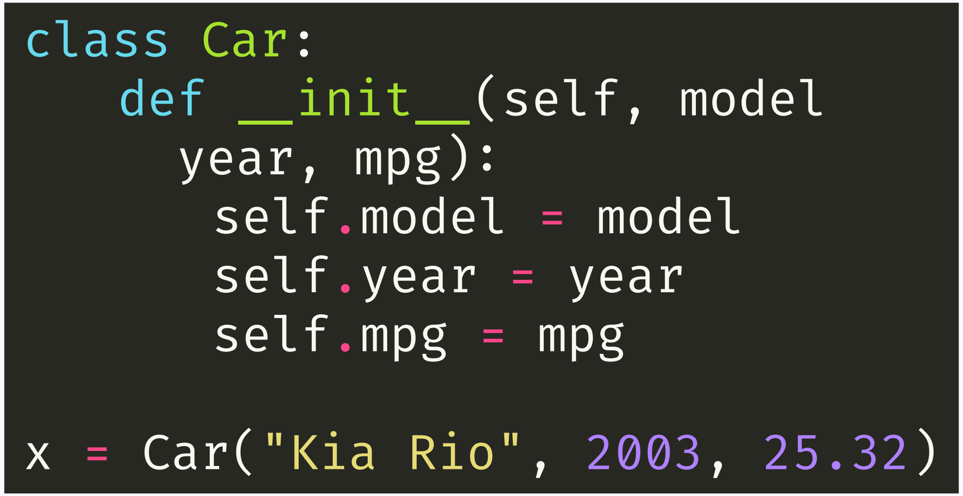
```
x.model = "Kia Soul"
```

Have different state with constructors

- Constructors allow passing in of data at object creation.
- This can allow for changing (and avoiding) default state.



```python
class Car:
    model = "Kia Rio"
    year = 2003
    mpg = 25.32

    def __init__(self, model,
      year, mpg):
        self.model = model
        self.year = year
        self.mpg = mpg
x = Car("Kia Rio", 2003, 25.32)
```

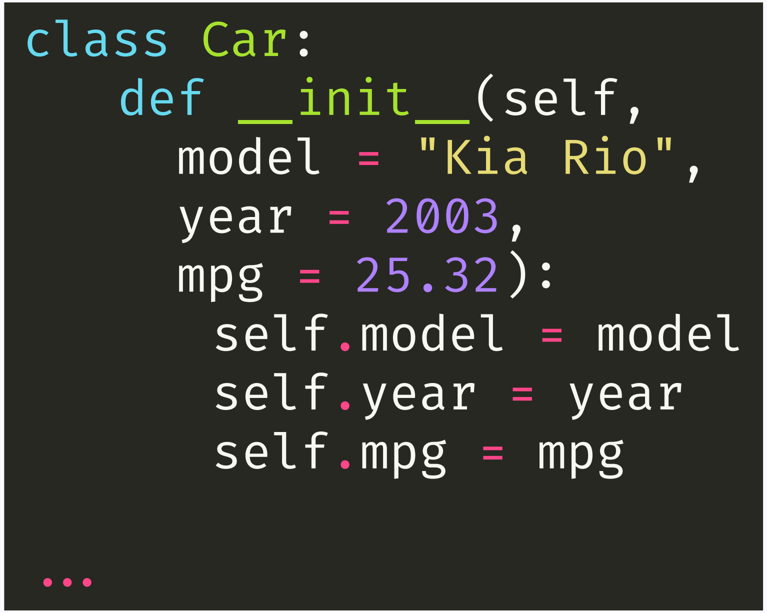


```python
class Car:
    def __init__(self, model
      year, mpg):
        self.model = model
        self.year = year
        self.mpg = mpg

x = Car("Kia Rio", 2003, 25.32)
```

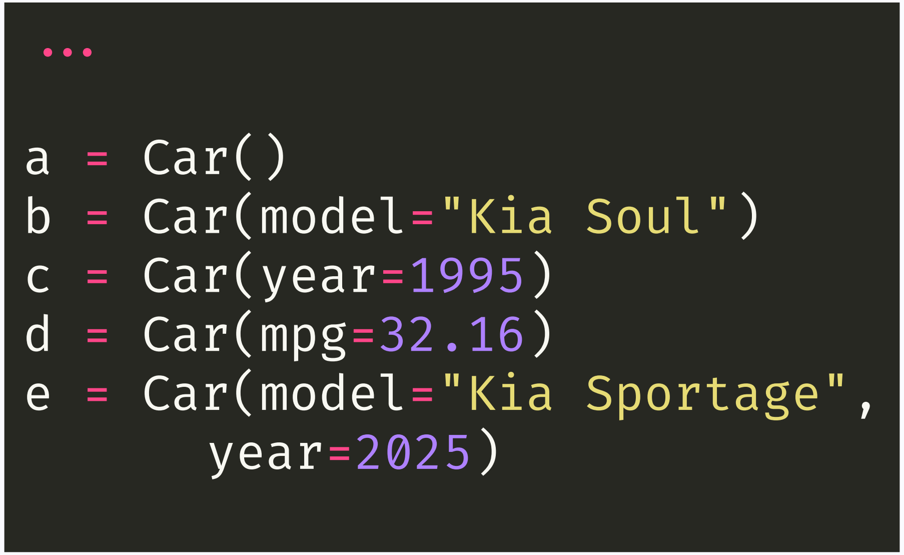
No Defaults

We can still have defaults!

- Simply set constructor’s arguments to optional.



```
class Car:
   def __init__(self,
     model = "Kia Rio",
     year = 2003,
     mpg = 25.32):
      self.model = model
      self.year = year
      self.mpg = mpg

...
```

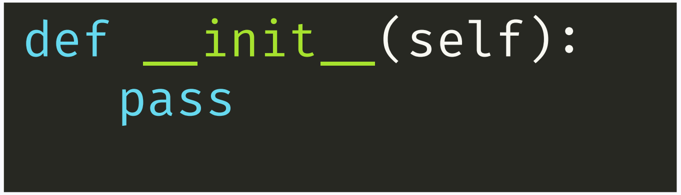


```
 ...

a = Car()
b = Car(model="Kia Soul")
c = Car(year=1995)
d = Car(mpg=32.16)
e = Car(model="Kia Sportage",
      year=2025)
```

What is the state of objects a, b, c, d, and e after creation?

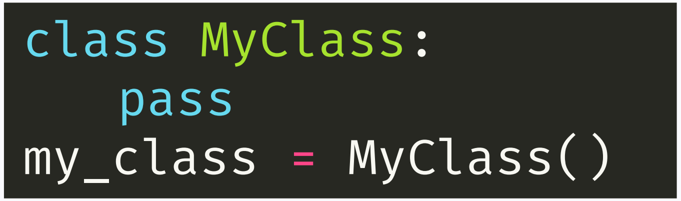
Every class has a constructor

- If you do not provide a constructor a default one is provided.



```
def __init__(self):
    pass
```

- Even without constructor you can still create objects which will call the default constructor.



```
class MyClass:
    pass
my_class = MyClass()
```

Printing looks weird

```
class Car:
    def __init__(self, model):
        self.model = model
x = Car("Honda")
print(x)
```

```
<__main__.Car object at 0x7fb8e5094ec0>
```

Create a **__str__** method.

```
class Car:
    def __init__(self, model):
        self.model = model
    def __str__(self):
        return f"Vehicle model:
           {self.model}"
x = Car("Honda")
print(x)
```

```
Vehicle model: Honda
```

Create methods to bind functionality

- Methods are functions written in a class.

```
class Car:
    def __init__(self, model):
        self.model = model

    def my_model(self):
        return self.model

    def car_sound():
        print("Vrooom")
```

```
x = Car("Ford")
print(x.my_model())
Car.car_sound()
print(Car.my_model(x))
```

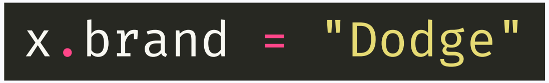
When calling a method on an object, it passes itself as first argument.

self

- Variable that references the current instance of the class (the object) to get associated variables.
- Not necessary *except* for __init__. Those without are called **static**.
- Can be called anything but must always be the first argument.

Working with objects

- Mutate attribute by assignment.



```
x.brand = "Dodge"
```

- Delete attribute with del.

```
del x.brand
```

- Delete object with del.

```
del x
```

Let’s practice

- Create a class **Stats** which has a constructor that takes in a list of numbers. Create the following methods:
  - **average**: which returns the average of the list.
  - **median**: which returns the median of the list.
  - **min:** which returns the minimum of the list.
  - **max:** which returns the maximum of the list.

Let’s practice (Continued)

- Create a class **Timer** with the following methods:
  - **start**: which starts the timer.
  - **stop**: which ends the timer.
  - **elapsed:** which returns the elapsed time.
  - **reset:** which resets the timer.
- You can get the current time with **time.time().**

OOP

The holy grail

“Everything is an object. Treat everything like an obj

Source: Monty Python and the Holy Grail

Pizza Restaurant

- Let’s say I am opening a pizza restaurant off Granville and West Georgia. What classes would I create to model this restaurant?

Source: Arnold Gatilao

Source: Wikimedia

Class for each object

- Generally, every physical entity should have its own class. E.g. a table, a car, a television, a shirt, etc.
- Similarly, non-physical entities should have their own class. E.g. a lecture, an idea, a law, a game, etc.

Any possible situation can be modelled as a group of objects with class definitions.

How objects relate

- If an object has another, use an attribute.
- If an object is/is like another, use inheritance.

Pizza

has

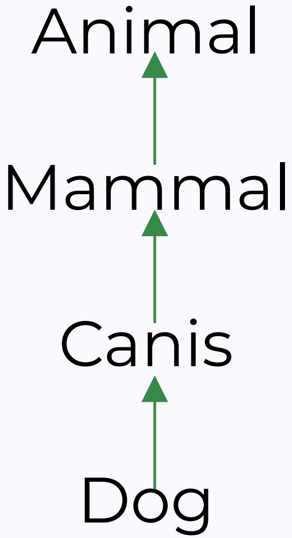
is like

Source: Eva K.

Source: Deryck Chan

Inheritance

- Inheritance is a way to save code by copying all the **methods and attributes** from the parent to the child.



Animal

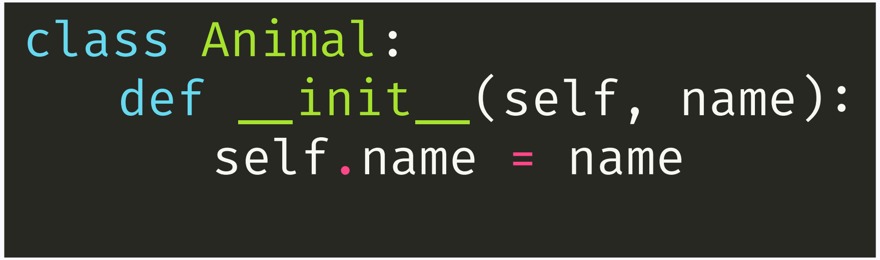
Mammal

Canis

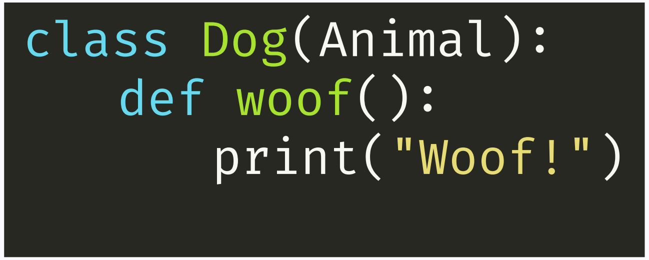
Dog

Inheriting

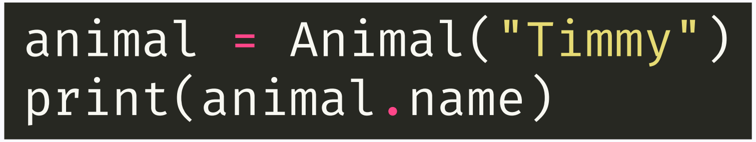
- You inherit a class by placing the name of the parent class in parentheses after the child’s class name declaration.



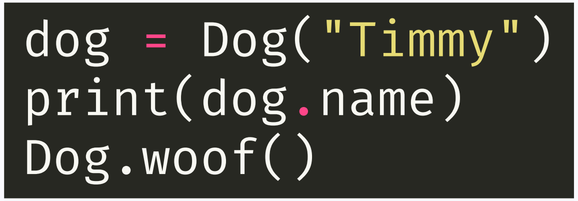
```python
class Animal:
    def __init__(self, name):
        self.name = name
```



```python
class Dog(Animal):
    def woof():
        print("Woof!")
```



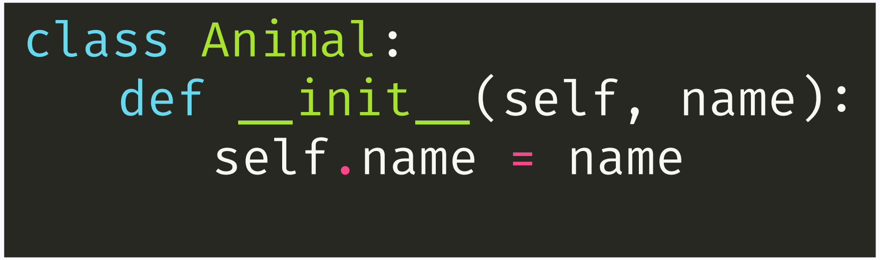
```python
animal = Animal("Timmy")
print(animal.name)
```



```python
dog = Dog("Timmy")
print(dog.name)
Dog.woof()
```

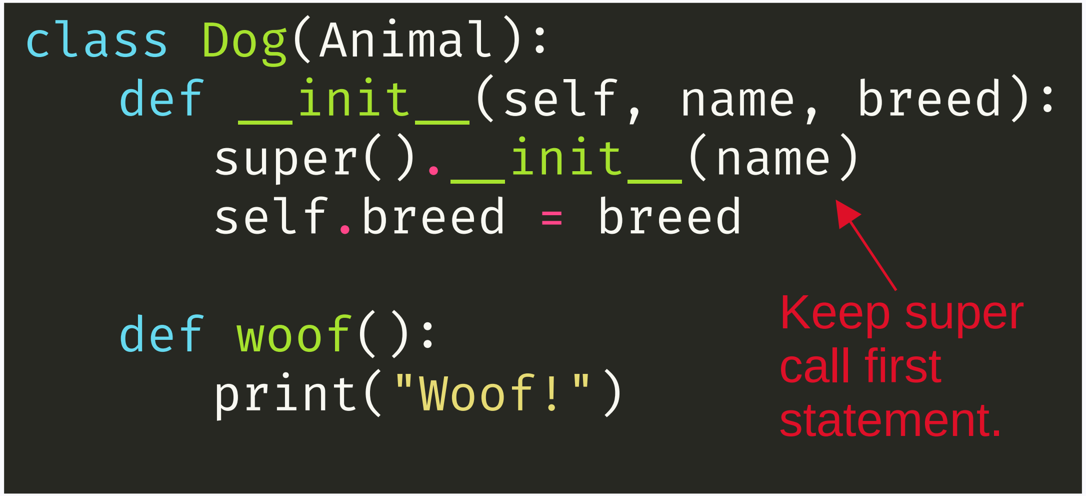
Not everything gets inherited

- Any methods that you define in the child's class that exist in the parent's class will not be inherited.



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

Only the method's name matters in determining whether to inherit or not.

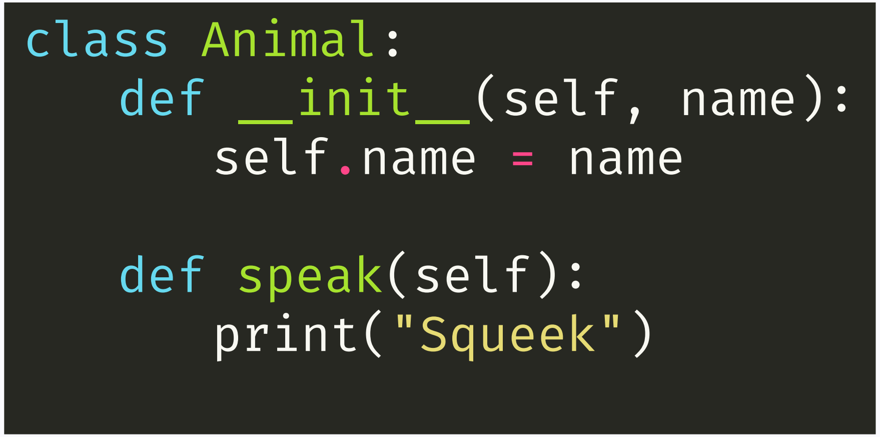


```python
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def woof():
        print("Woof!")
```

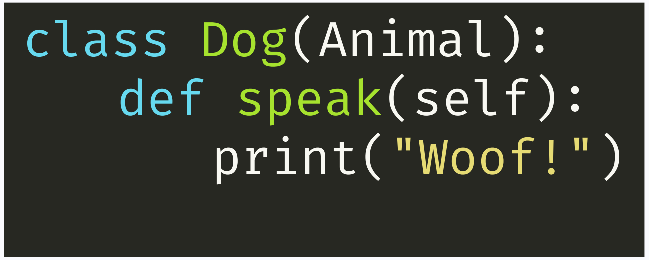
Keep super call first statement.

Inheritance can make objects behave differently

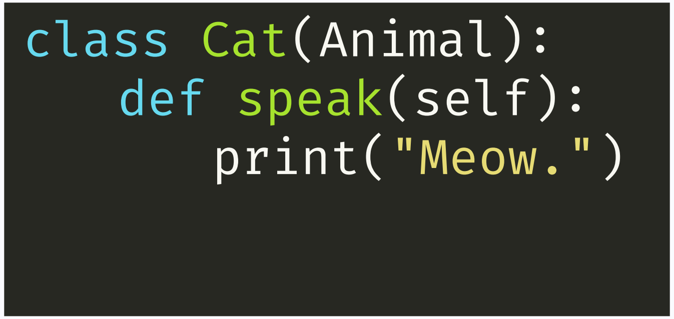
- Two objects may have a common superclass with a common method which behaves differently (Polymorphism).



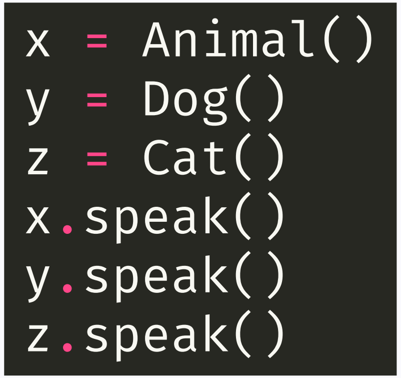
```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Squeek")
```



```python
class Dog(Animal):
    def speak(self):
        print("Woof!")
```



```python
class Cat(Animal):
    def speak(self):
        print("Meow.")
```



```python
x = Animal()
y = Dog()
z = Cat()
x.speak()
y.speak()
z.speak()
```