Python Programming: Objects and Classes

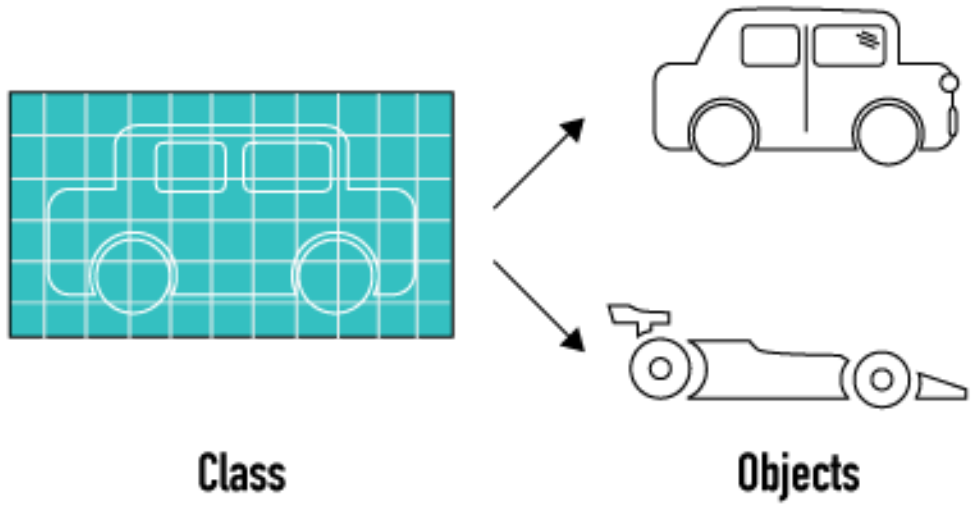
Learning Objectives

After this lesson, you will be able to…

- Define a class.
- Instantiate an object from a class.
- Create classes with default instance variables.

Blueprints

All cars have things that make them a `Car`. Although the details might be different, every type of car has the same basics — it's off the same blueprint, with the same properties and actions.



- Property: A shape (could be hatchback or sedan).
- Property: A color (could be red, black, blue, or silver).
- Property: Seats (could be between 2 and 5).
- Action: Can drive.
- Action: Can park.

Introduction: Objects and Classes

These properties and behaviors can be thought of as variables and functions.

`Car` blueprint:

- Properties (variables): `shape`, `color`, `seats`
- Actions (functions): `drive()` and `park()`

An actual car might have:

```
# **Properties - Variables**:
- shape = "hatchback"
- color = "red", "black", "blue", or "silver"
- seats = 2


# **Actions - Functions:**
- drive()
- park()
- reverse()
```

Discussion: What might a blueprint for a chair look like?

Discussion: Python Classes

In Python, the concept of blueprints and objects is common. A **class** is the blueprint for an **object**. Everything you declare in Python is an object, and it has a class.

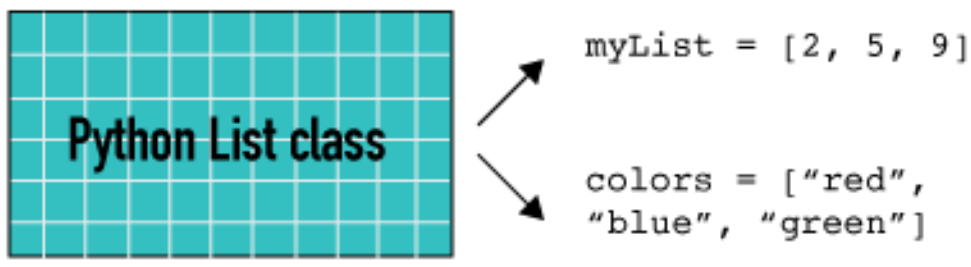
Consider the `List` class — every list you make has the same basic concept.

Variables:

- Elements: What's in the list! E.g., `my_list = [element1, element2, element3]`.

Functions that all lists have:

- `my_list.pop()`, `my_list.append()`, `my.list.insert(index)`



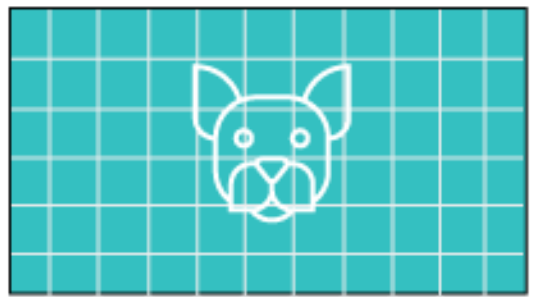
What behaviors and properties do you think are in the `Dictionary` class? The `Set` class?

Discussion: A `Dog` Class

We can make a class for anything! Let's create a `Dog` class.

The objects might be `greyhound`, `goldenRetriever`, `corgi`, etc.

Think about the `Dog` blueprint. What variables might our class have? What functions?



Pro tip: When functions are in a class, they are called "methods." They're the same thing!

Pro tip: While objects are named in snake_case, classes are conventionally named in TitleCase.

We Do: Defining Classes

Follow along! Let's create a new file, `Dog.py`.

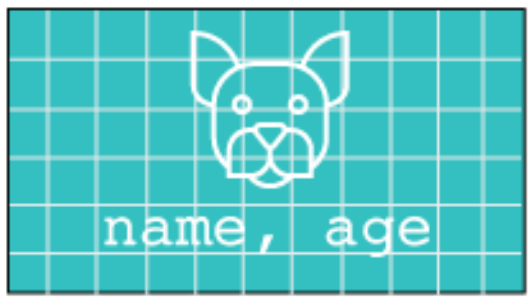
Class definitions are similar to function definitions, but instead of `def`, we use `class`.

Let's declare a class for `Dog`:

```
class Dog:

 # We'll define the class here.

 # Our dog will have two variables: name and age.
```



Pro tip: Files are usually named for their class, so the `Dog` class is in `Dog.py`.

We Do: Adding Docstrings, Part 1

Using our `Dog` class from the previous exercise, let's add a *docstring*.

A docstring is a comment that is placed at the top of a class (or function). When python precompiles the class or function (which happens in the background), it parses this comment and makes it available to the user as a helpfile when the class is accessed when typing the class or function, followed by a `?` question mark. In a Jupyter

Notebook, this can also be accessed by placing your cursor at the end of a class or function and pressing `SHIFT + TAB`. Try it out! Enter `list?` in a python interpreter or Jupyter cell and run it. What is returned?

```
In [1]: list?
Init signature: list(self, /, *args, **kwargs)
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
Type:           type
```

Here, we can see under the `Docstring` section are the 'instructions for use', and the `Init signature` is a list of arguments that can be passed to the function, which is automatically recognized and generated by the python compiler.

We Do: Adding Docstrings, Part 2

Using our `Dog` class from the previous exercise, let's add a *docstring*.

The docstring for a function or method should summarize its behavior and document its arguments, return value(s), side effects, exceptions raised, and restrictions on when it can be called. Not all of these will be applicable for every function or method, so the programmer should take care to document the parts relevant. Optional arguments should be indicated. It should be documented whether keyword arguments are part of the interface.

The first line is a simple description of the class or function, followed by a blank line, followed by keyword arguments, if any.

Let's write a docstring for `Dog`:

```
class Dog:
    """Creates Dog class, possible child class of Animal


    Parameters
    ----------
    name : str, default blank
        Desired name of Dog
    age : int, default 0
```

We Do: The `__init__` Method

What first? Every class starts with an `__init__` method. It's:

- Where we define the class' variables.
- Short for "initialize."
  - "Every time you make an object from this class, do what's in here."

Let's add this:

```
class Dog:
    """Creates Dog class, possible child class of Animal


    Parameters
    ----------
    name : str, default blank
        Desired name of Dog
    age : int, default 0
        Age of Dog in years
    """
```

We Do: Adding a `bark_hello()` Method

All dogs have the behavior `bark`, so let's add that. This is a regular function (method), just inside the class!

```
class Dog:


    def __init__(self, name="", age=0):
    # Note the defaults.

      self.name = name # All dogs have a name.
      self.age = age # All dogs have an age.


    # All dogs have a bark function.
    """Prints a message stating name and age to stdout"""
    def bark_hello(self):
```

We're done defining the class!

Note: We have also added a docstring to our new `.bark_hello()` method. This will be omitted for future examples for brevity.

Aside: Instantiating Objects From Classes

Now we have a `Dog` template!

Each `dog` object we make from this template:

- Has a name.
- Has an age.
- Can bark.

We Do: How Do We Make a `Dog` Object?

We call our class name like we call a function — passing in arguments, which go to the `init`.

Add this under your class (non-indented!):

```
# Declare the objects.
gracie = Dog("Gracie", 8)
spitz = Dog("Spitz", 5)
buck = Dog("Buck", 3)


# Test them out!
gracie.bark_hello()
print("This dog's name is", gracie.name)
print("This dog's age is", gracie.age)
spitz.bark_hello()
buck.bark_hello()
```

Try it! Run `Dog.py` like a normal Python file: `python Dog.py`.

We Do: Adding Print

`__init__` is just a method. It creates variables, but we can also add a `print` statement! This will run when we create the object.

```
class Dog:

  def __init__(self, name="", age=0):
    self.name = name
    self.age = age
    print(name, "created.") # Run when init is finished.


  def bark_hello(self):
    print("Woof! I am called", self.name, "; I am", self.age, "human-years c


fox = Dog("Fox") #  Note that "Fox created." prints — and we're using the de
fox.bark_hello()
```

Try it!

Quick Review: Classes

A class is a blueprint for an object. Some classes are built into Python, like `List`. We can always make a `list` object.

We can make a class for anything!

```
# Created like a function; TitleCase
class Dog:


  # __init__: A method (function) that happens just once, when the object is
  def __init__(self, name="", age=0): # What's passed in to the class is use
    # Set variables for each.
    self.name = name
    self.age = age
    print(name, "created.") # This will run when the __init__ method is call


# Classes can have as many methods (functions) as you'd like.
```

Discussion: What About Tea?

Let's make a `TeaCup` class.

- What variables would a cup of tea have?
- What methods?

A Potential `TeaCup` Class

We could say:

Variables:

- A total `capacity`.
- A current `amount`.

Methods:

- `fill()` our cup.
- `empty()` our cup.
- `drink()` some tea from our cup.

Example: A `TeaCup` Class

Here's what a `TeaCup` class definition might look like in Python:

```
class TeaCup:
  def __init__(self, capacity):
    # Python executes when a new cup of tea is created.
    self.capacity = capacity # Total ounces the cup holds.
    self.amount = 0 # Current ounces in the cup. All cups start empty!


def fill(self):
  self.amount = self.capacity


def empty(self):
  self.amount = 0
```

Quick Knowledge Check:

```
class TeaCup:
  def __init__(self, capacity = 8):
    self.capacity = capacity
    self.amount = 0
```

When will the capacity be `8`?

Variables for All Class Objects

Next up: new types of class variables!

Let's revisit our `Dog` class:

```
class Dog:


  def __init__(self, name="", age=0):
    self.name = name
    self.age = age
    print(name, "created.")


def bark_hello(self):
  print("Woof! I am called", self.name, "; I am", self.age, "human-years c
```

What if there are variables that we want across all dogs?

For example, can we count how many `dog` objects we make and track it in the class?

I Do: Class vs. Instance Members

We already have **instance variables**, which are specific to each `dog` object (each has its own name!).

A **class variable** is specific to the class, regardless of the object. It's created **above** `__init__`.

```
class Dog:


  ### Here, we define class variables. ###

  # These are the same for ALL dogs.

  total_dogs = 0


  def __init__(self, name="", age=0):


    ### These are instance variables. ###

    self.name = name

    self.age = age
```

I Do: Tallying Dogs

We can increment the `class` variable any time.

```
class Dog:
  total_dogs = 0
  def __init__(self, name="", age=0):
    self.name = name
    self.age = age
    Dog.total_dogs += 1 #  We can increment this here!
    print(name, "created:")

def bark_hello(self):
  print("Woof! I am called", self.name, "; I am", self.age, "human-years c
  print("There are", Dog.total_dogs, "dogs in this room!")
```

Partner Exercise: Create a Music Genre Class

Pair up! Create a new file, `Band.py`.

- Define a class, `Band`, with these instance variables: `"genre"`, `"band_name"`, and `"albums_released"` (defaulting to `0`).
- Give `Band` a method called `print_stats()`, which prints a string like `"The rock band Queen has 15 albums."`
- Create a class variable, `number_of_bands`, that tracks the number of bands created.

Test your code with calls like:

```
my_band = ("Queen", 15, "rock")
```

Bonus: If the genre provided isn't `"pop"`, `"classical"`, or `"rock"`, print out `"This isn't a genre I know."`

Partner Exercise: Create a `BankAccount` Class

Switch drivers! Create a new class (and file), `Bank.py`.

Bank accounts should:

- Be created with the `accountType` property (either `"savings"` or `"checking"`).
- Keep track of its current `balance`, which always starts at `0`.
- Have access to `deposit()` and `withdraw()` methods, which take in an integer and update `balance` accordingly.
- Have a class-level variable tracking the total amount of money in all accounts, adding or subtracting whenever `balance` changes.

Bonus: Start each account with an additional `overdraftFees` property that begins at `0`. If a call to `withdraw()` ends with the `balance` below `0`, then `overdraftFees` should be incremented by `20`.

Knowledge Check: Select the Best Answer

Consider the following class definition for `Cat`:

```
class Cat:
 def __init__(self, name='Lucky'):
  self.name = name
  self.fur = short
```

How would you instantiate a `Cat` object with the `name` attribute `'Furball'`?

1. `mycat = Cat(name='Furball')`
2. `furball = Cat`
3. `mycat = Cat(self, name='Furball')`
4. `mycat = Cat.init(name='Furball')`

Knowledge Check: Select All That Apply.

Which of the following statements are true about the `self` argument in class definitions?

- The user does not need to supply `self` when using instance methods.
- The `self` argument is a reference to the instance object.
- Any variable assigned with `self` (e.g., `self.var`) will be shared across instances of the class.
- With an instance object, `obj`, entering `obj.self.var` will provide the value for `var` for that instance.

Knowledge Check: Select the Best Answer

Consider the following code:

```
class Shape(object):
 possible = ['triangle','square','circle','pentagon','polygon','rectangle']


 def __init__(self, label='triangle'):
  self.label = label


def is_possible(self):
 if self.label in self.possible:
  print('This is possible')
 else:
  print('This is impossible')
```

If you were to enter `wormhole.is_possible()`, would the outcome be `"This is possible"` or `"This is impossible"`?

Summary: Discussion

Let's chat! Can anyone explain:

- What a class is?
- What `__init__` does?
- What an object is?
- The point of `self`?
- The two types of variables?

Summary and Q&A

Class:

- A pre-defined structure that contains attributes and behaviors grouped together.
- The blueprint.
- Defined via a method call.
- Contains an `__init__` method that takes in parameters to be assigned to an object.
- E.g., the `Dog` class; the `List` class.

Object:

- An instance of a class structure.
- The items built from the blueprint.
- E.g., the `gracie` object; the `my_list` object.

Summary: Types of Variables in a Class

Instance variables:

- Contain data types declared in the class but defined in each object.
- Each `dog` has its own name and age.
- Each `my_list` has its own elements.

Class variables:

- Contain data and actions that span across all objects.
- How many `dog` objects are there in total?
- The `self` keyword lets us distinguish between variables that exist at the class level versus in each object.