Beginner's Python Cheat Sheet - Classes

What are classes?

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs such as dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects.

Classes can inherit from each other—you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations. Even if you don't write many of your own classes, you'll frequently find yourself working with classes that others have written.

Creating and using a class

Consider how we might model a car. What information would we associate with a car, and what behavior would it have? The information is assigned to variables called attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.

The Car class

```
class Car:
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

Creating and using a class (cont.)

Creating an instance from a class

```
my_car = Car('audi', 'a4', 2021)
```

Accessing attribute values

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```
my_car.fill_tank()
my_car.drive()
```

Creating multiple instances

```
my_car = Car('audi', 'a4', 2024)
my_old_car = Car('subaru', 'outback', 2018)
my_truck = Car('toyota', 'tacoma', 2020)
my_old_truck = Car('ford', 'ranger', 1999)
```

Modifying attributes

You can modify an attribute's value directly, or you can write methods that manage updating values more carefully. Methods like these can help validate the kinds of changes that are being made to an attribute.

Modifying an attribute directly

```
my_new_car = Car('audi', 'a4', 2024)
my_new_car.fuel_level = 5
```

Writing a method to update an attribute's value

```
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

Writing a method to increment an attribute's value

```
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
            <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

Naming conventions

In Python class names are usually written in CamelCase and object names are written in lowercase with underscores. Modules that contain classes should be named in lowercase with underscores.

Class inheritance

If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to introduce new attributes and methods, and override attributes and methods of the parent class.

To inherit from another class include the name of the parent class in parentheses when defining the new class.

The __init__() method for a child class

```
class ElectricCar(Car):
    """A simple model of an electric car."""

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attributes specific to electric cars.
        # Battery capacity in kWh.
        self.battery_size = 40

        # Charge level in %.
        self.charge_level = 0
```

Adding new methods to the child class

```
class ElectricCar(Car):
    --snip--

    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

Using child methods and parent methods

```
my_ecar = ElectricCar('nissan', 'leaf', 2024)

my_ecar.charge()
my_ecar.drive()
```

Finding your workflow

There are many ways to model real world objects and situations in code, and sometimes that variety can feel overwhelming. Pick an approach and try it; if your first attempt doesn't work, try a different approach.

Python Crash Course

A Hands-on, Project-Based Introduction to Programming

ehmatthes.github.io/pcc_3e

Class inheritance (cont.)

Overriding parent methods

```
class ElectricCar(Car):
    --snip--

    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

Instances as attributes

A class can have objects as attributes. This allows classes to work together to model more complex real-world things and concepts.

A Battery class

```
class Battery:
    """A battery for an electric car."""

    def __init__(self, size=85):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 40:
            return 150
        elif self.size == 65:
            return 225
```

Using an instance as an attribute

```
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```
my_ecar = ElectricCar('nissan', 'leaf', 2024)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

Importing classes

Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered, you can store your classes in modules and import the classes you need into your main program.

Storing classes in a file

car.py

```
"""Represent gas and electric cars."""

class Car:
    """A simple attempt to model a car."""
    --snip—

class Battery:
    """A battery for an electric car."""
    --snip--

class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

Importing individual classes from a module

my_cars.py

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2021)
my_beetle.fill_tank()
my_beetle.drive()

my_leaf = ElectricCar('nissan', 'leaf', 2024)
my_leaf.charge()
my_leaf.drive()
```

Importing an entire module

```
import car

my_beetle = car.Car(
        'volkswagen', 'beetle', 2021)
my_beetle.fill_tank()
my_beetle.drive()

my_leaf = car.ElectricCar('nissan', 'leaf',
        2024)
my_leaf.charge()
my_leaf.drive()
```

Importing all classes from a module

(Don't do this, but recognize it when you see it.)

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2021)
my_leaf = ElectricCar('nissan', 'leaf', 2024)
```

Storing objects in a list

A list can hold as many items as you want, so you can make a large number of objects from a class and store them in a list.

Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.

A fleet of rental cars

```
from car import Car, ElectricCar

# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []

# Make 250 gas cars and 500 electric cars.
for _ in range(250):
    car = Car('ford', 'escape', 2024)
    gas_fleet.append(car)
for _ in range(500):
    ecar = ElectricCar('nissan', 'leaf', 2024)
    electric_fleet.append(ecar)

# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print(f"Gas cars: {len(gas_fleet)}")
print(f"Electric cars: {len(electric_fleet)}")
```

Understanding self

People often ask what the `self` variable represents. The `self` variable is a reference to an object that's been created from the class.

The `self` variable provides a way to make other variables and objects available everywhere in a class. The `self` variable is automatically passed to each method that's called through an object, which is why you see it listed first in most method definitions. Any variable attached to `self` is available everywhere in the class.

Understanding `__init__()`

The `__init__()` method is a function that's part of a class, just like any other method. The only special thing about `__init__()` is that it's called automatically every time you make a new instance from a class. If you accidentally misspell `__init__()`, the method won't be called and your object may not be created correctly.