# Notes on Object Orientated Programming

**Last updated Spring 2020 by Jemmy Zhou**

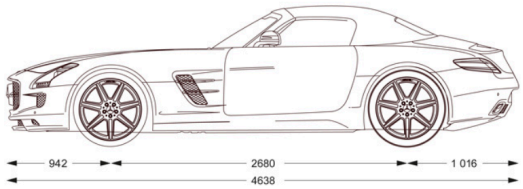
*Disclaimer: These are not official class notes. They're just meant to be a quick reference. Please let me know if there are any typos or mistakes.*

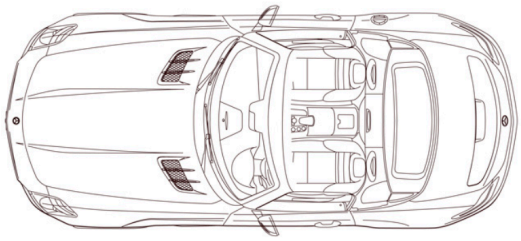
## 1. Classes

Goal: Implement some kind of racing (cars) game. To do so, we will need to make cars! Well how do we define these cars? Let's take a deeper look. 🤓

Rather than implementing each cars as a list (as we did with ADTs), let's think of each car in the game as an individual object. Well, if we want to create these cars, we must have some kind of blueprint! Unlucky for us, downloading an exisiting one off the Internet will probably get us fined so we'll have to make our own.

I'm a big fan of Merecedes, so our first blueprint can be for a Mercedes SLS AMG! Let's think about what goes into making this specific car.



First, let's list some properties that all cars of this specific model will have:

- 4 wheels
- 2 doors
- make = "Mercedes"
- model = "SLS AMG"

Okay, now that we have the general idea for this blueprint, let's start designing in some customer specifics. Let's say I want to buy this specific one:

What are some properties that are specific to this one that I really, really want?

- color = "red"
- license = "UWU"

Ok, looks like we have a pretty good idea of what our blueprints going to look like! To formalize what we are doing, we're transitioning into the mindset of **Object Orientated Programming (OOP)**.

## Key Definitions:

- This idea of designing a blueprint is what programmers call designing a **class**.
- We use classes to model the **objects** we plan on making in the world we're trying to create.
- Creating objects of a class is also known as making **instances of a class**.
- We call properties that are shared among all instances of a class **class attributes**.
- On the other hand, we call properties that are unique to each instance of a class **instance attributes**.

A simple class definition will look like the following:

```
class <CLASS_NAME>:
    <CLASS_ATTRIBUTE>
    <CLASS_ATTRIBUTE>
    ...
    def __init__(self):
```

Wait! What is this function doing here? And where do I put my instance attributes? This will be easier to explain with a concrete example, so I've defined the Mercedes SLS AMG class below.

```
class Mercedes_SLS_AMG():
    make = 'Mercedes'
    model = 'SLS AMG'
    wheels = 4
    doors = 2
```

If this is the blueprint I have so far, then I can expect that every Mercedes SLS AMG vehicle I create will have those 4 properties. Well now I want my custom order with red paint and the license plate "UWU". So let's trying creating one.

But wait! In order to instantiate (create an instance of) a class, I need an init method (a.k.a. function; we'll henceforth call all functions inside a class methods, for reasons to be detailed below), hence `def __init__(self)`. **Almost every class you define will have this initializing method.** This is the method where you pass in special properties pertaining to your object and bind them to your specific instance (hmm… properties unique to your object, sound familiar?).

Let's add an init method to `Mercedes_SLS_AMG`.

```
class Mercedes_SLS_AMG():
    make = 'Mercedes'
    model = 'SLS AMG'
    wheels = 4
    doors = 2

    def __init__(self, license, color):
        self.license = license
        self.color = color
```

To create my specific car, the correct notation is

```
>>> jemmys_mercedes = Mercedes_SLS_AMG('UWU', 'red')
```

To create an instance of a class, we call the class name on the parameters of the `__init__` method. We exclude the `self` argument because that is the object being created! Therefore, the first argument we pass in actually matches to the second argument of the `__init__` method. We'll explore how the instance attributes are being defined in the following section.

## 2. Attributes

Recall that **class attributes** are properties that belong to every instance of a class. Since these variables don't belong to any object in particular we can define them just like typical variables `attr = value`.

On the other hand, **instance attributes** are properties that are unique to each instance of a class. Thus, we must "bind" each property to its respective instance. Notice how this can only be achieved when we can actually access the object. As of now, the only time we can access this object is either in the `__init__` method (referenced by `self`) or in the frame after we define the object, which is `jemmys_mercedes` in the example.

Let's take a look at what is going on in the `__init__` method.In the example, I call `Mercedes_SLS_AMG('UWU', 'red')`. This passes in the String `'UWU'` to the argument `license` and the string `'red'` to the argument 'color'. Next, we evaluate the assignment statement `self.license = license`. We evaluate this assignment statement just like any other we've seen. The right hand side evalutes to `'UWU'` in the local frame.

The left hand side is what we call **dot notation**. Whenever we have `<instance>.<attribute>`, we can think of this as the `<attribute>` belonging to `<instance>`. In this case, we create a new attribute specific to `self` called `license` with a value of `'UWU'`. This is how we define instance attributes. **All instance attributes are**

**defined in the form `<object>.<attribute>`**. In the following line, we define another instance attribute `color` with value `red`.

Thus, we can slightly generalize the earlier statement. **All instance attributes should be defined inside a method using the `self` argument**. Although it is allowed, rarely do we define instance attributes after we make the object.

Here's an example:

```
>>> jemmys_mercedes = Mercedes_SLS_AMG('UWU', 'red')
>>> jemmys_mercedes.make    # class attribute
'Mercedes'
>>> jemmys_mercedes.color   # instance attribute
'red'
>>> jemmys_mercedes.is_cool = True      # Don't do this
```

How do we know when to look for an instance attribute and when to look for a class attribute? Simple, just follow these 3 steps:

1. Look at the object's instance attributes. If found, return. Else:
2. Look at the object's class attributes. If found, return. Else:
3. Error

*(we'll expand upon this in the Inheritance section)*

Below, I've summarized the difference between instance attributes and class attributes.

1. **Instance Attributes**
    - Defined inside methods (typically)
    - Property of the instance (unique for each instance)
    - Notation for defining is `self.attr_name = value` (inside a method)
    - Notation for referencing is `self.attr_name` (inside a method)
2. **Class Attributes**
    - Defined outside of methods (generally above)
    - Property of the class (same for every instance)
    - Notation for defining is `attr_name = value`
    - Notation for referencing is `CLASS.attr_name` or `self.attr_name`
        - NOTE: Latter only works if there is no instance attribute with the same `attr_name`.

In general,

- We **CANNOT** reference attributes as just `attr_name`.
- Instances can have instance attributes with the same name as class attributes. Python will "override" the class attribute with the instance attribute.

# 3. Functions vs. Methods

We've seen the `__iter__` method already, so let's generalize this to all other methods. Methods are really just functions inside a class. I've outlined the similarities and differences below:

- Similarities:
  - Executes one line at a time
  - Returns some result (can be None)
- Differences:
  - Functions take in 0 or more parameter(s).
  - Methods take in 1 or more parameter(s). Why at least 1?

There are a couple of special methods (Python calls them **magic methods**). Why magic? Because we aren't really interested in understanding exactly how they work, e.g. `__init__`, we just need to know what they do. Here are some notes:

- What each magic method does/returns is unique
  - For the car example, when we create Jemmy's, i.e. `Mercedes_SLS_AMG('UWU', 'red')`, Python will magically create `self`, and pass in `'UWU'` as the second argument and `'red'` as the third. We then execute the `__init__` method line-by-line and then `__init__` returns the object we created. How does it do it? We don't know. But at least we know what it does 😄
- Some other magic methods:
  - `__str__(self)`
  - `__repr__(self)`
  - `__iter__(self)`
  - and many more!

# 4. Method Calls

1. Either `self.method(<params>)` or `CLASS.method(self, <params>)` work.
2. When invoking `self.method(<params>)`, the instance `self` is implicity passed in as the first parameter.
3. When invoking `CLASS.method(<params>)`, we have to explicity pass in `self` as the first parameter.

# 5. `__str__` vs `__repr__`

Please refer to page 6 of the worksheet for the specific definitions. Below are just some notes I've compiled:

- Both `__str__` and `__repr__` magic methods **always** return strings
- Every class created has default `__str__` and `__repr__` magic methods.
- The default `__repr__` method will return something like `<__main__.CLASSNAME object ...>`. You don't need to worry about what any of that means, you'll likely only be asked to write `object` in those situations.

- The default `__str__` method will call `__repr__` and return that.
- `print(object)` will do the following:
    - call `str(object)` which redirects to `object.__str__`
    - return a string
    - strip the quotes & display
- Evaluating an `object` will do the following:
    - call `repr(object)` which redirects to `object.__repr__`
    - return a string
    - strip the quotes & display

strip quotes = remove outter most quotes

# 6. Inheritance

Suppose we want a broader class of Mercedes vehicles. We can define the class `Mercedes` to be:

```
class Mercedes:
    make = 'Mercedes'
    wheels = 4
    doors = 4

    def __init__(self, license, color):
        self.license = license
        self.color = color
```

Now, let's redefine the `Mercedes_SLS_AMG` class to inherit from the `Mercedes` class. The correct notation for inheritance is:

```
class Mercedes_SLS_AMG(Mercedes):
    model = 'SLS_AMG'
    doors = 2
```

When a **subclass** (think child) inherits from a **super class** (think parent), we don't need to rewrite anything that is shared between the two. An analogy is, for the first few years of your life, there is no need to go to work and make your own money for rent/food. Your parent already has it, so you can just use it. However, if there are any differences between the subclass and super class, then we must override the pre-exisiting definitions in the super class.

Looking back at the car example, we do not need to define an `__init__` method in `Mercedes_SLS_AMG` because we can just inherit the one from `Mercedes`–it works fine. However, we have 2 fewer doors than a typical `Mercedes`, so we have to override the variable.

General Notes:

1. The subclass inherits everything the super class has:
    - class attributes
    - methods (both regular and magic)

    - instance attributes (defined in the methods)
2. The subclass can improve on what the super class already does:
    - *i.e. Overriding a method from the super class*
    - When overriding, the method in the subclass class has to have the **exact same name** as the method in the super class.
3. The subclass can do new things:
    - *i.e. Define new methods, new class attributes, new instance attributes…*
    - Can invoke methods of the super class by calling `SUPER_CLASS.method(self, ...)`

As a final note, we can update the lookup for an attribute to be:

1. Look at the object's instance attributes. If found, return. Else:
2. Look at the class attributes. If found, return. Else:
3. Look at the parent's class attributes. If found, return. Else:
4. Repeat step 3 until there is no more parent. Then Error.