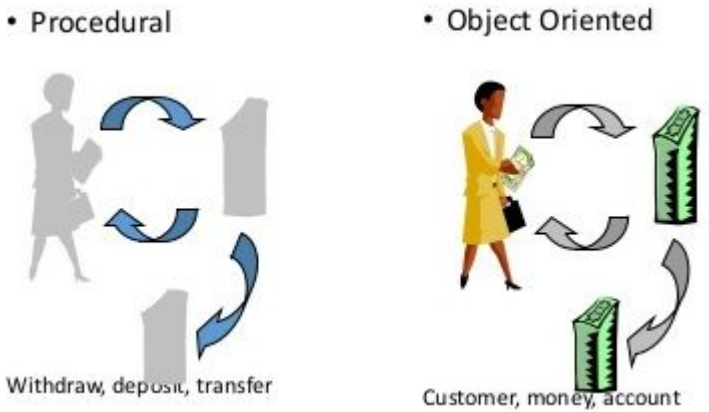
OOP

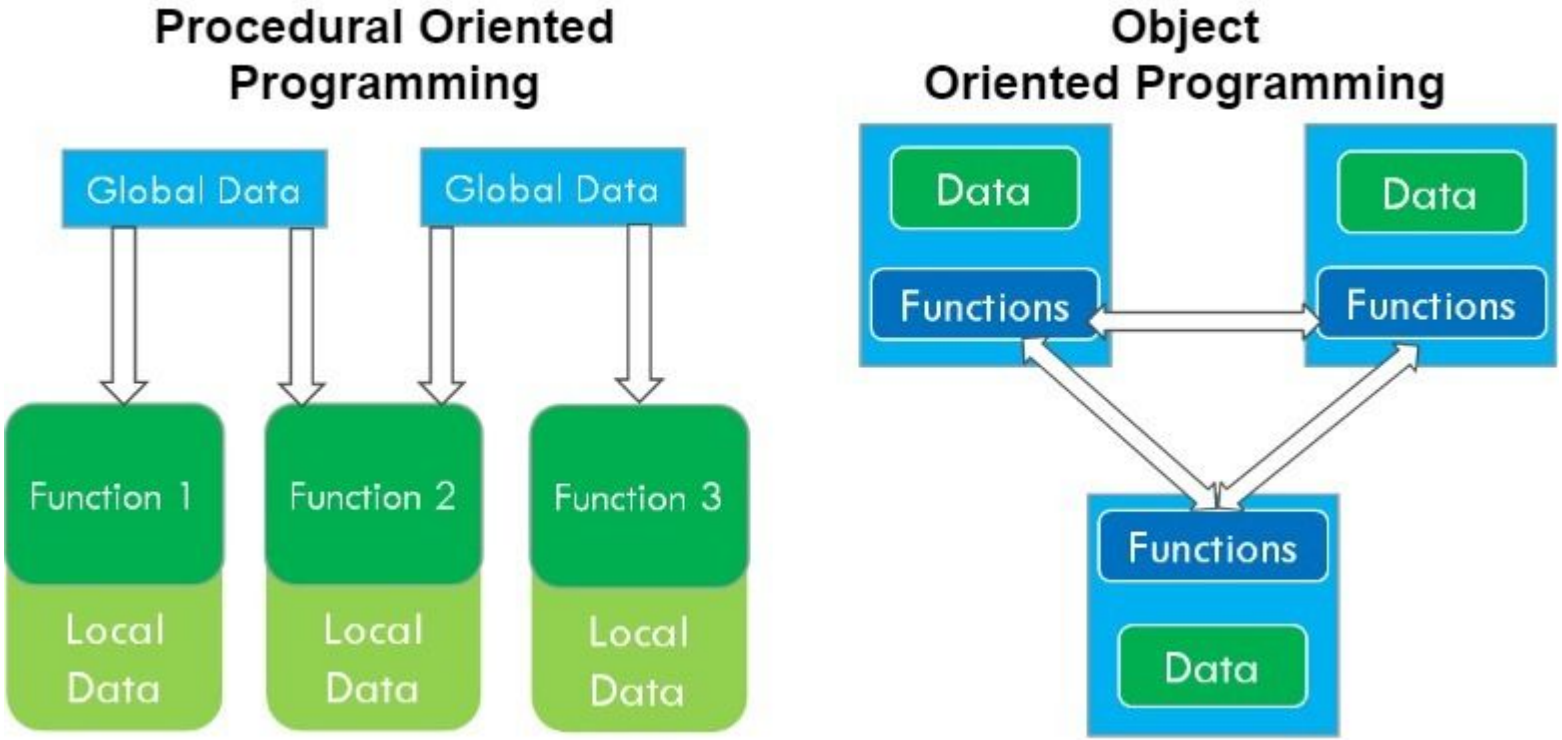
Object-oriented programming
part. 1

OOP vs Procedural

- OOP - models real-world entities as software objects.
- Procedural programing structures a program like a recipe where a set of steps is provided.



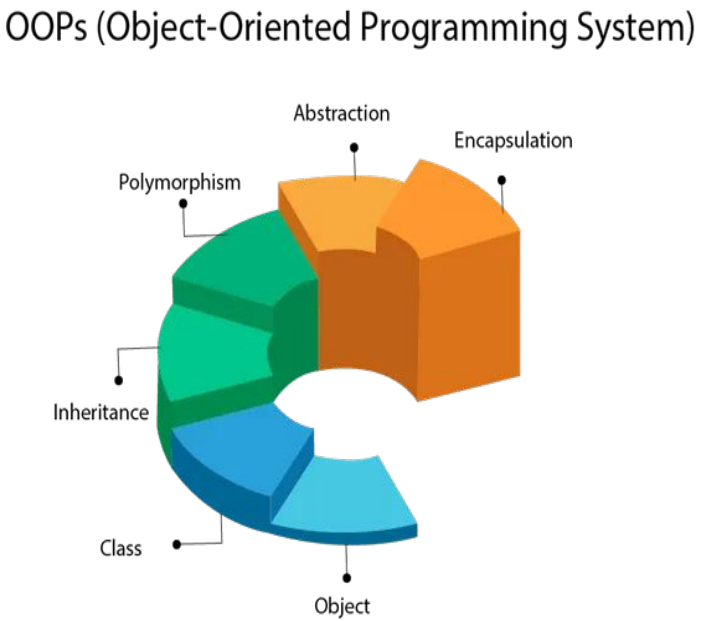
OOP vs Procedural



Object Oriented Programming: A curated set of resources

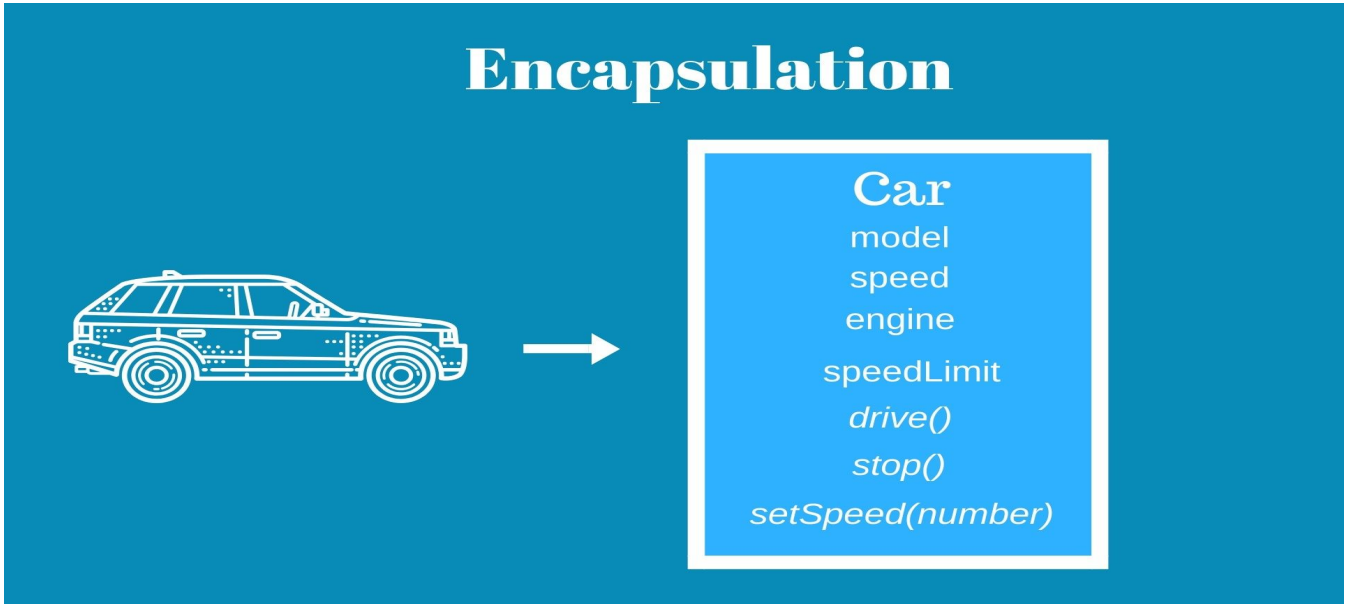
Object-Oriented Programming or OOP is a particular way of programming that leverage the concept of Classes and Objects and the following 4 paradigms:

- **Abstraction**: a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.
- **Encapsulation**: a practice that binds the data with the code that manipulates it and keeps the data and the code safe from external interference
- **Inheritance**: the mechanism by which an object acquires the some (or all) properties of another object.
- **Polymorphism**: a way to process objects differently based on their data type. In other words, objects can have the same name for a method but the implementation may differ.



Encapsulation

Encapsulation is an approach for restricting direct access to some of the data structure elements (fields, properties, methods, etc).



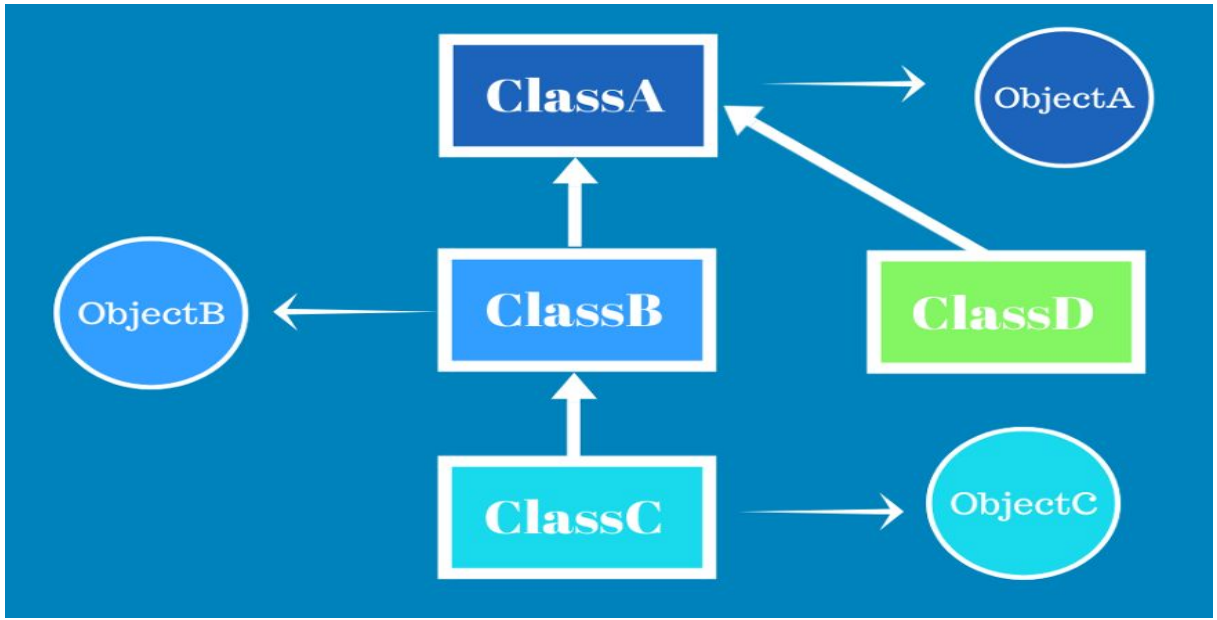
Encapsulation: Attribute types

1. public - can be freely used inside or outside of a class definition.
2. protected - should not be used outside of class definition
3. private - inaccessible and invisible.

```
1  class Robot:
2      def __init__(self, name=None):
3          self.__name = name
4
5      # setter for private attr name
6      def set_name(self, name):
7          self.__name = name
8
9      # getter for private attr name
10     def get_name(self):
11         return self.__name
12
13 x = Robot("Marvin")
14 print(x.get_name())
```

Inheritance

Inheritance is an approach of sharing common functionality within a collection of classes. It provides an ability to avoid code duplication in a class that needs the same data and functions which another class already has.



Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

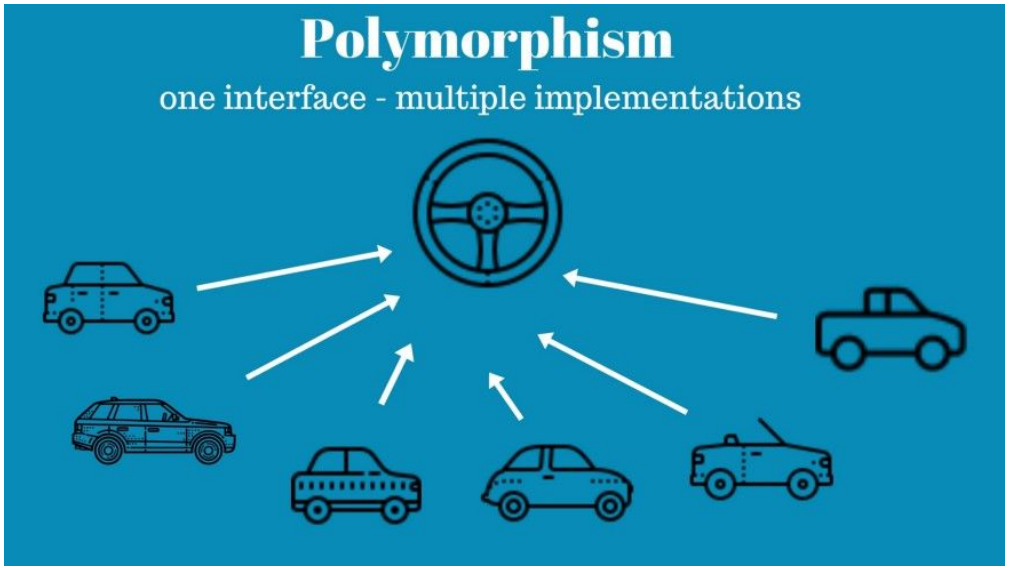
- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def print_name(self):
        print(self.first_name, self.last_name)


john = Person("John", "Barbbie")
john.print_name()
```

```
class Student(Person):
    pass


mike = Student("Mike", "Torse")
mike.print_name()
```

Polymorphism

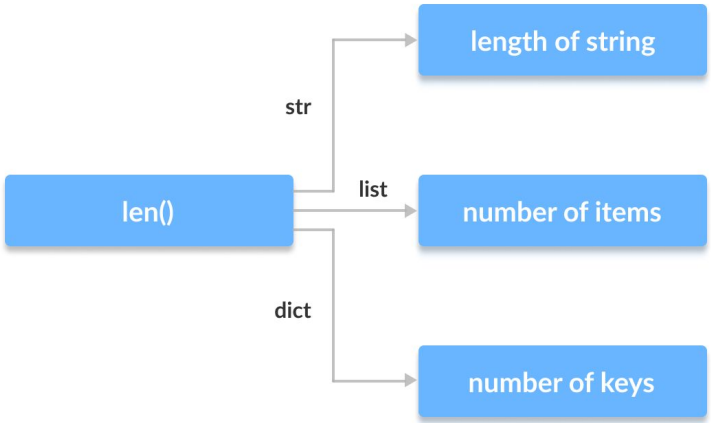
Polymorphism is an ability to substitute classes that have common functionality in sense of methods and data. In other words, it is an ability of multiple object types to implement the same functionality that can work in a different way but supports a common interface.



Polymorphism

Polymorphic len() function:

```
print(len("Programiz"))

print(len(["Python", "Java", "C"]))

print(len({"Name": "John", "Address": "Nepal"}))
```



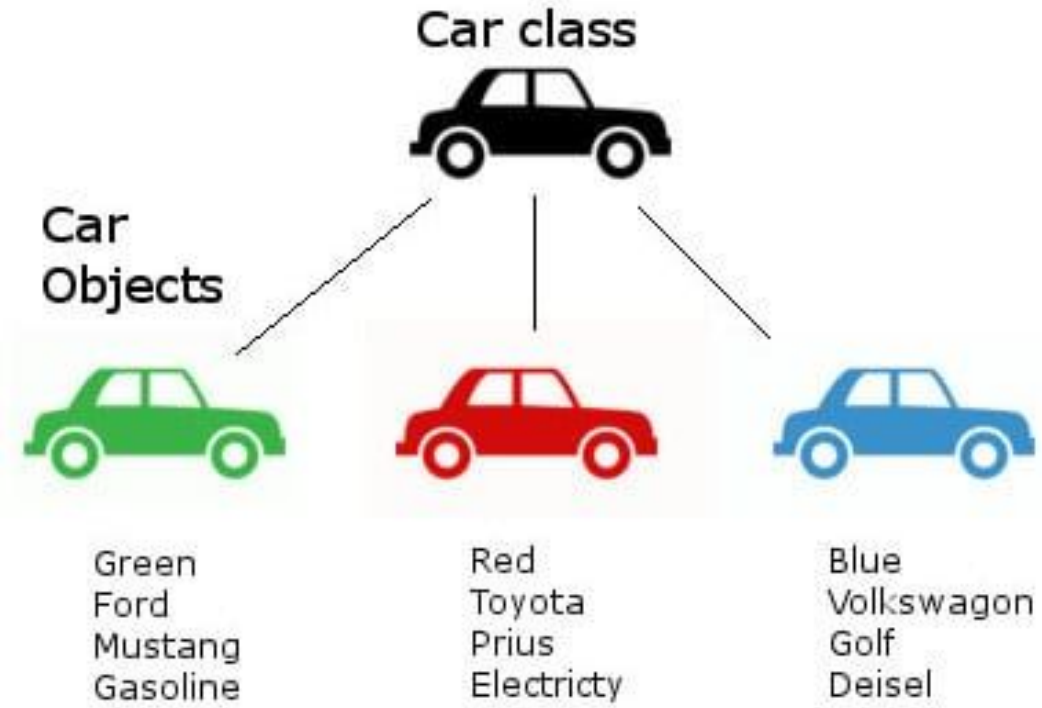
Main Principles

DRY (Don't Repeat Yourself). - The concept of OOP in Python focuses on creating reusable code.

KISS (keep it simple stupid) - The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.

YAGNI (You aren't gonna need it) - is a principle of extreme programming (XP) that states a programmer should not add functionality until deemed necessary.

Object concept



Instance of class

A class is a blueprint for the object. It defines **attributes** and **methods.** Attribute is a variable stored in an instance or class. Method is a function stored in an instance or class.

Creating object of class/ instantiating class.

Accessing class/instance attributes

```
class Car:  # class keyword is used to define class Car.
    car_type = 'regular car'

    def __init__(self, color, brand, model, engine_type):
        self.color = color
        self.brand = brand
        self.model = model
        self.engine_type = engine_type


car_1 = Car('Green', 'Ford', 'Mustang', 'Gasoline')
car_2 = Car('Red', 'Toyota', 'Prius', 'Electricity')
car_3 = Car('Blue', 'VW', 'Golf', 'Diesel')
# access the class attributes
print(f"Car 1 is a {car_1.__class__.car_type}")
print(f"Car 2 is also a {car_2.__class__.car_type}")
print(f"Car 3 is also a {car_3.__class__.car_type}")
# access the instance attributes
print(f"{car_1.brand} is {car_1.color}")
print(f"{car_2.brand} is {car_2.color}")
print(f"{car_3.brand} is {car_3.color}")
```

Inheritance. Built-in functions

Python has two **built-in functions** that work with inheritance:

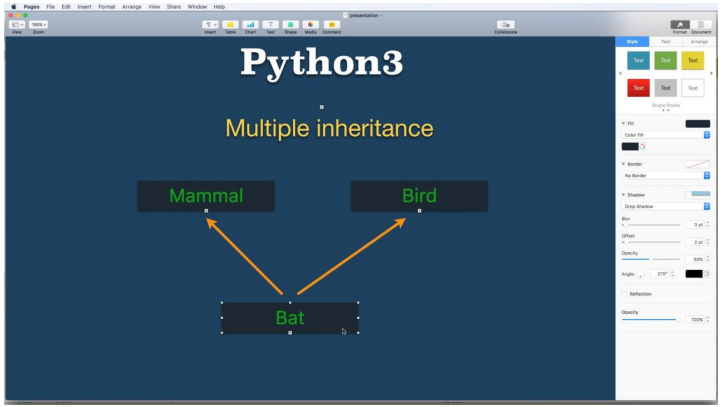
Use **isinstance()** to check an instance's type: isinstance(obj, int) will be True only if obj.__class__ is int or some class derived from int.

Use **issubclass()** to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int. However, issubclass(float, int) is False since float is not a subclass of int.

Multiple Inheritance

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in DerivedClassName, it is searched for in Base1, then (recursively) in the base classes of Base1, and if it was not found there, it was searched for in Base2, and so on.

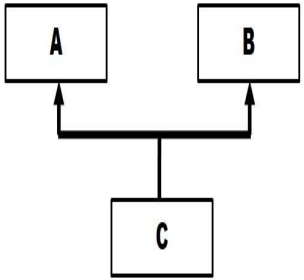
```
class DerivedClassName(Base1,
Base2,Base3):
    <statement-1>
    <statement-2>
    <statement-..>
    <statement-N>
```



Method Resolution Order (MRO)

MRO is the order in which Python looks for a method in a hierarchy of classes. Especially it plays vital role in the context of multiple inheritance as single method may be found in multiple super classes.

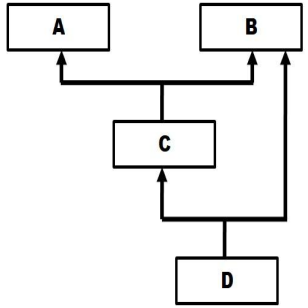
```
class A:
    def process(self):
        print('A process()')


class B:
    def process(self):
        print('B process()')

class C(A, B):
    pass

obj = C()
obj.process()
print(C.mro())
```



Output:

```
A process()
[<class '__main__.C'>, <class
'__main__.A'>, <class
'__main__.B'>, <class 'object'>]
```

```
class A:
    def process(self):
        print('A process()')

class B:
    def process(self):
        print('B process()')

class C(A, B):
    def process(self):
        print('C process()')

class D(C,B):
    pass

obj = D()
obj.process()
print(D.mro())
```



Output:

```
C process()
[<class '__main__.D'>, <class
'__main__.C'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'object'>]
```

Magic Methods

Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action. For example, when you add two numbers using the + operator, internally, the **__add__()** method will be called.

Built-in classes in Python define many magic methods. Use the **dir()** function to see the number of magic methods inherited by a class.

Magic Methods

List of important magic methods:

- __init__() - initialization method, to get called by the __new__ method
- __new__() - returns a new object, which is then initialized by __init__()
- __del__() - destructor method.
- __call__() - allows instances of our to be callable
- __str__() - return a printable string representation of any user defined class
- __repr__() - return a machine readable representation of a type
- __dir__() - return a list of attributes of a class
- __getattr__(self, name) - Is called when the accessing attribute of a class that does not exist.
- __setattr__(self, name, value) - Is called when assigning a value to the attribute of a class
- __delattr__(self, name) - Is called when deleting an attribute of a class
- __iter__ - returned in a number of contexts
- __next__ - to get called by the __iter__

Construction and Initialization

__new__(cls, [...)
__new__ is the first method to get called in an object's instantiation. It takes the class, then any other arguments that it will pass along to __init__. __new__ is used fairly rarely, but it does have its purposes.

__init__(self, [...)
The initializer for the class. It gets passed whatever the primary constructor was called with (so, for example, if we called x = SomeClass(10, 'foo'), __init__ would get passed 10 and 'foo' as arguments. __init__ is almost universally used in Python class definitions.

__del__(self)
If __new__ and __init__ formed the constructor of the object, __del__ is the destructor. It doesn't implement behavior for the statement del x (so that code would not translate to x.__del__()). Rather, it defines behavior for when an object is garbage collected.

Comparison magic methods

`__cmp__(self, other)`
The most basic of the comparison magic methods. It actually implements behavior for all of the comparison operators (<, ==, !=, etc.),

`__eq__(self, other)`
Defines behavior for the equality operator, `==`.

`__ne__(self, other)`
Defines behavior for the inequality operator, `!=`.

`__lt__(self, other)`
Defines behavior for the less-than operator, `<`.

`__gt__(self, other)`
Defines behavior for the greater-than operator, `>`.

`__le__(self, other)`
Defines behavior for the less-than-or-equal-to operator, `<=`.

`__ge__(self, other)`
Defines behavior for the greater-than-or-equal-to operator, `>=`.

Callable Objects

`__call__(self, [args...])`
Allows an instance of a class to be called as a function. Essentially, this means that `x()` is the same as `x.__call__()`. Note that `__call__` takes a variable number of arguments; this means that you define `__call__` as you would any other function, taking however many arguments you'd like it to.

```python
class Entity:
    """Class to represent an entity. Callable to update the entity's position."""

    def __init__(self, size, x, y):
        self.x, self.y = x, y
        self.size = size

    def __call__(self, x, y):
        """Change the position of the entity."""
        self.x, self.y = x, y
```

Representing your Classes

__str__(self)
Defines behavior for when str() is called on an instance of your class.

__repr__(self)
Defines behavior for when repr() is called on an instance of your class. The major difference between str() and repr() is intended audience. repr() is intended to produce output that is mostly machine-readable (in many cases, it could be valid Python code even), whereas str() is intended to be human-readable.

__dir__(self)
Defines behavior for when dir() is called on an instance of your class. This method should return a list of attributes for the user. Typically, implementing __dir__ is unnecessary, but it can be vitally important for interactive use of your classes if you redefine __getattr__ or __getattribute__ (which you will see in the next section) or are otherwise dynamically generating attributes.