Self in Python, Demystified

Even when we understand the use of self, it may still seem odd, especially to programmers coming from other languages, that self is passed as a parameter explicitly every single time we define a method. As **The Zen of Python** goes, "**Explicit is better than implicit**".

Self represents the instance of the class. By using 'self' we can access the attributes and method of the class in python. It binds the attributes with the given arguments.

The reason you need to use self is because python does not @syntax to refer to instance attributes. Python decided not to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.

That makes methods entirely the same as functions, and leaves the actual name to use up to you (although self is the convention, and most developers will generally frown at you when you use something else.) self is not special to the code, it's just another object.

Python could have done something else to distinguish normal names from attributes -- special syntax like Ruby has, or requiring declarations like C++ and Java do, or perhaps something yet more different -- but it didn't. Python's all for making things explicit, making it obvious what's what, and although it doesn't do it entirely everywhere, it does do it for instance attributes. That's why assigning to an instance attribute needs to know what instance to assign to, and that's why it needs self.
Self must be provided as a first parameter to the instance method and constructor. If you don't provide it, it will cause an error.

```
# Sample Python Code

class car():

        # init method or constructor
        def __init__(self, model, color):
                self.model = model
                self.color = color

        def show(self):
                print("Model is", self.model )
                print("color is", self.color )

# both objects have different self which
# contain their attributes
audi = car("audi a4", "blue")
ferrari = car("ferrari 488", "green")

audi.show()      # same output as car.show(audi)
ferrari.show() # same output as car.show(ferrari)

#note:we can also do like this
print("Model for audi is ",audi.model)
print("Colour for ferrari is ",ferrari.color)
#this happens because after assigning in the constructor the attributes are linked to that particular object
#here attributes(model,colour) are linked to objects(audi,ferrari) as we initialize them
# Behind the scene, in every instance method
# call, python sends the instances also with
# that method call like car.show(audi)
```

You might have seen `__init__()` very often but the use of `__new__()` is rare. This is because most of the time you don't need to override it. Generally, `__init__()` is used to initialize a newly created object while `__new__()` is used to control the way an object is created.
We can also use `__new__()` to initialize attributes of an object, but logically it should be inside `__init__()`.
One practical use of `__new__()`, however, could be to restrict the number of objects created from a class.

All the best,
Isaac Muuo

Bibliography

[1] https://www.geeksforgeeks.org/self-in-python-class/.

[2] https://www.programiz.com/article/python-self-why.