The Story of self Parameter in Python, Demystified

If you have been programming in Python (in object oriented way of course) for some time, I'm sure you have come across methods that have **self** as their first parameter. It may seem odd, especially to programmers coming from other languages, that this is done explicitly every single time we define a method. As **The Zen of Python** goes, "**Explicit is better than implicit**".

So, why do we need to do this? Let's take a simple example to begin with. We have a **Point** class which defines a method **distance** to calculate the distance from origin.

```
class Point(object):
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y

    def distance(self):
        """Find distance from origin"""
        return (self.x**2 + self.y**2) ** 0.5
```

Let us now instantiate this class and find the distance.

```
>>> p1 = Point(6,8)
>>> p1.distance()
10.0
```

In the above example, **__init__()** defines three parameters but we just passed two (6 and 8). Similarly **distance()** requires one but zero arguments were passed. Why is Python not complaining about this argument number mismatch?

What Happens Internally?

Let me first clarify that **Point.distance** and **p1.distance** in the above example are a bit different.

```
>>> type(Point.distance)
<class 'function'>
>>> type(p1.distance)
<class 'method'>
```

We can see that the first one is a function and second, a method. A peculiar thing about methods (in Python) is that the object itself is passed on as the first argument to the corresponding function. In the case of the above example, the method call **p1.distance()** is actually equivalent to **Point.distance(p1)**. Generally, when we call a method with some arguments, the corresponding class function is called by placing the method's object before the first argument. So, anything like **obj.meth(args)** becomes **Class.meth(obj, args)**. The calling process is automatic while the receiving process is not (its explicit).

This is the reason the first parameter of a function in class must be the object itself. Writing this parameter as **self** is merely a convention. It is not a keyword and has no special meaning in Python. We could use other names (like **this**) but I strongly suggest you not to. Using names other than **self** is frowned upon by most developers and degrades the readability of the code ("**Readability counts**").

Self Can Be Avoided

By now you are clear that the object (instance) itself is passed along as the first argument, automatically. This implicit behavior can be avoided by making a method, **static**. Consider the following simple example.

```
class A(object):

    @staticmethod
    def stat_meth():
        print("Look no self was passed")
```

Here, **@staticmethod** is a function decorator which makes **stat_meth()** static. Let us instantiate this class and call the method.

```
>>> a = A()
>>> a.stat_meth()
Look no self was passed
```

From the above example, we are clear that the implicit behavior of passing the object as the first argument was avoided using static method. All in all, static methods behave like our plain old functions.

```
>>> type(A.stat_meth)
<class 'function'>
>>> type(a.stat_meth)
<class 'function'>
```

Self Is Here To Stay

The explicit **self** is not unique to Python. This idea was borrowed from **Modula-3**. Following is a use case where it becomes helpful.

There is no explicit variable declaration in Python. They spring into action on the first assignment. The use of **self** makes it easier to distinguish between instance attributes (and methods) from local variables. In, the first example **self.x** is an instance attribute whereas **x** is a local variable. They are not the same and lie in different namespaces.

Many have proposed to make self a keyword in Python, like **this** in C++ and Java. This would eliminate the redundant use of explicit **self** from the formal parameter list in methods. While this idea seems promising, it's not going to happen. At least not in the near future. The main reason is backward compatibility. Here is a blog from the creator of Python himself explaining why the explicit self has to stay.

__init__() is not a constructor

One important conclusion that can be drawn from the information so far is that, **__init__()** is not a constructor. Many naive Python programmers get confused with it since **__init__()** gets called when we create an object. A closer inspection will reveal that the first parameter in **__init__()** is the object itself (object already exists). The function **__init__()** is called immediately **after** the object is created and is used to initialize it.

Technically speaking, constructor is a method which creates the object itself. In Python, this method is **__new__()**. A common signature of this method is

```
__new__(cls, *args, **kwargs)
```

When **__new__()** is called, the class itself is passed as the first argument automatically. This is what the **cls** in above signature is for. Again, like **self**, **cls** is just a naming convention. Furthermore, ***args** and ****kwargs** are used to take arbitary number of arguments during method calls in Python.

Some important things to remember when implementing __new__() are:

- **__new__()** is always called before **__init__()**.
- First argument is the class itself which is passed implicitly.
- Always return a valid object from **__new__()**. Not mandatory, but thats the whole

point.

Let's take a look at an example to be crystal clear.

```
class Point(object):

    def __new__(cls,*args,**kwargs):
        print("From new")
        print(cls)
        print(args)
        print(kwargs)

        # create our object and return it
        obj = super().__new__(cls)
        return obj

    def __init__(self,x = 0,y = 0):
        print("From init")
        self.x = x
        self.y = y
```

Now, let's now instantiate it.

```
>>> p2 = Point(3,4)
From new
<class '__main__.Point'>
(3, 4)
{}
From init
```

This example illustrates that __new__() is called before __init__(). We can also see that the parameter cls in __new__() is the class itself (Point). Finally, the object is created by calling the __new__() method on object base class. In Python, object is the base class from which all other classes are derived. In the above example, we have done this using super().

Use __new__ or __init__?

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.

Suppose we wanted a class SqPoint for creating instances to represent the four vertices of a square. We can inherit from our previous class Point (first example in this article) and use __new__() to implement this restriction. Here is an example to restrict a class to have only four instances.

```
class SqPoint(Point):
    MAX_Inst = 4
    Inst_created = 0

    def __new__(cls,*args,**kwargs):
        if (cls.Inst_created >= cls.MAX_Inst):
            raise ValueError("Cannot create more objects")
        cls.Inst_created += 1
        return super().__new__(cls)
```

A sample run.

```
>>> p1 = SqPoint(0,0)
>>> p2 = SqPoint(1,0)
>>> p3 = SqPoint(1,1)
>>> p4 = SqPoint(0,1)
>>>
>>> p5 = SqPoint(2,2)
Traceback (most recent call last):
...
ValueError: Cannot create more objects
```