`Self`

08 March 2023 15:19

Q: What is the purpose of the `self` parameter? Why is it needed?

Explanation 1:

Let's say you have a class ClassA which contains a method methodA defined as:

```
def methodA(self, arg1, arg2):
  # do something
```

and objectA is an instance of this class.

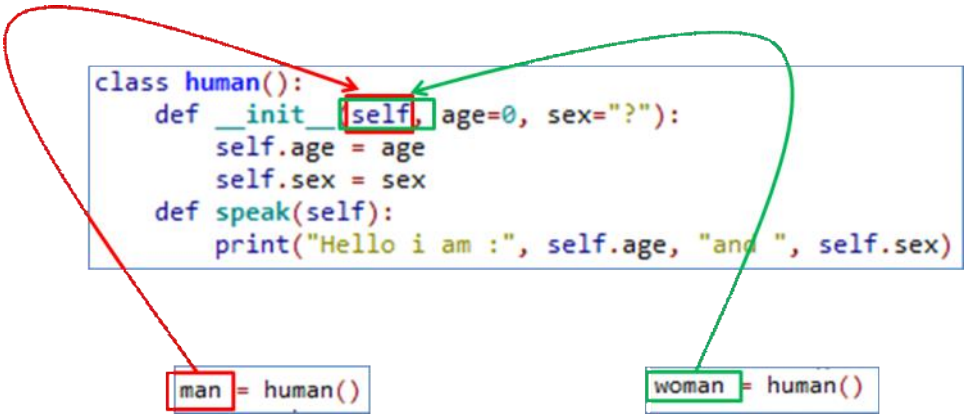
Now when objectA.methodA(arg1, arg2) is called, python internally converts it for you as:

```
ClassA.methodA(objectA, arg1, arg2)
```

The `self` variable refers to the object itself.

Explanation 2:

When objects are instantiated, the object itself is passed into the `self` parameter.



Because of this, the object's data is bound to the object. Below is an example of how you might like to visualize what each object's data might look. Notice how 'self' is replaced with the objects name. I'm not saying this example diagram below is wholly accurate but it hopefully with serve a purpose in visualizing the use of self.

```
class human():
    def __init__(man, age=0, sex="?"):
        man.age = age
        man.sex = sex
    def speak(man):
        print("Hello i am :", man.age, "and ", man.sex)
```

```
class human():
    def __init__(woman, age=0, sex="?"):
        woman.age = age
        woman.sex = sex
    def speak(woman):
        print("Hello i am :", woman.age, "and ", woman.sex)
```

The Object is passed into the `self` parameter so that the object can keep hold of its own data. Although this may not be wholly accurate, think of the process of instantiating an object like this: When an object is made it uses the class as a template for its own data and methods. Without passing its own name into the `self` parameter, the attributes and methods in the class would remain as a general template and would not be referenced to (belong to) the object. So by passing the object's name into the `self` parameter it means that if 100 objects are instantiated from the one class, they can all keep track of their own data and methods.

See the illustration below:

The code below shows the instantiation of 2 objects and the result of accessing their individual data.

Bob's data being passed into the constructor method at the moment of instantiation.

Kate's data being passed into the constructor method at the moment of instantiation.

```
class Human():

    def __init__(self, name="no name", age=0,
                 nationalilty="no nationalilty", gender="no gender"):

        self.name = name
        self.age = age
        self.nationality = nationalilty
        self.gender = gender

    def speak(self):
        print("Hello, my name is: ", self.name)


bob = Human("Bob", 42, "British", "Male")
kate = Human("Kate", 22, "American", "Female")

print("Bob's Attributes:")
print("**********")
print(bob.name)
print(bob.age)
print(bob.nationality)
print(bob.gender)
print("")
print("Bob's Methods:")
print("**********")
bob.speak()
print("")
print("**********")
print("**********")
print("")
print("Kate's Attributes:")
print("**********")
print(kate.name)
print(kate.age)
print(kate.nationality)
print(kate.gender)
print("")
print("Kate's Methods:")
print("**********")
kate.speak()
```

```
Bob's Attributes:
**********
Bob
42
British
Male

Bob's Methods:
**********
Hello, my name is:  Bob

**********
**********

Kate's Attributes:
**********
Kate
22
American
Female

Kate's Methods:
**********
Hello, my name is:  Kate
```

In the code, you can see when we access the objects' data, each object has its own individual data. It manages to keep track of it.

This is due to the 'self.' prefix.