Understanding Object Instantiation and Metaclasses in Python

By Rupesh Mishra on Nov 25, 2020

In this article, we will cover the object instantiation process followed by Python internally to create objects. I'll start with the fundamentals of object creation, and then we'll dive deep into understanding specific methods, such as `__new__`, `__init__`, and `__call__`. We will also understand the `Metaclass` in Python, along with its role in the object creation process. Although these are advanced topics, the article explains each topic step-by-step and from scratch so that even beginners can understand it.

Please note that this article is written with Python3 in mind.

Table of Contents

The `object` base class in Python3

In Python3, all classes implicitly inherit from the built-in `object` base class. The `object` class provides some common methods, such as `__init__`, `__str__`, and `__new__`, that can be overridden by the child class. Consider the code below, for example:

```
class Human:
    pass
```

In the above code, the `Human` class does not define any attributes or methods. However, by default, the `Human` class inherits the `object` base class and as a result it has all the attributes and methods defined by the `object` base class. We can check all the attributes and the methods inherited or defined by the `Human` class using the `dir` function.

> The `dir` function returns a list of all the attributes and methods defined on any Python object.

```
dir(Human)

# Output:
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__'
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__'
```

The *dir* function's output shows that the *Human* class has lots of methods and attributes, most of which are available to the *Human* class from the *object* base class. Python provides a *__bases__* attribute on each class that can be used to obtain a list of classes the given class inherits.

> The *__bases__* property of the class contains a list of all the base classes that the given class inherits.

```
print(Human.__bases__)
# Output: (<class 'object'>,)
```

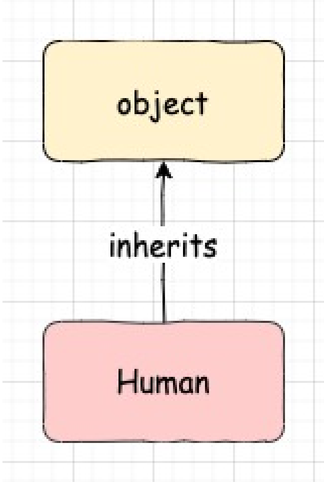
The above output shows that the *Human* class has *object* as a base class. We can also look at the attributes and methods defined by the *object* class using the *dir* function.

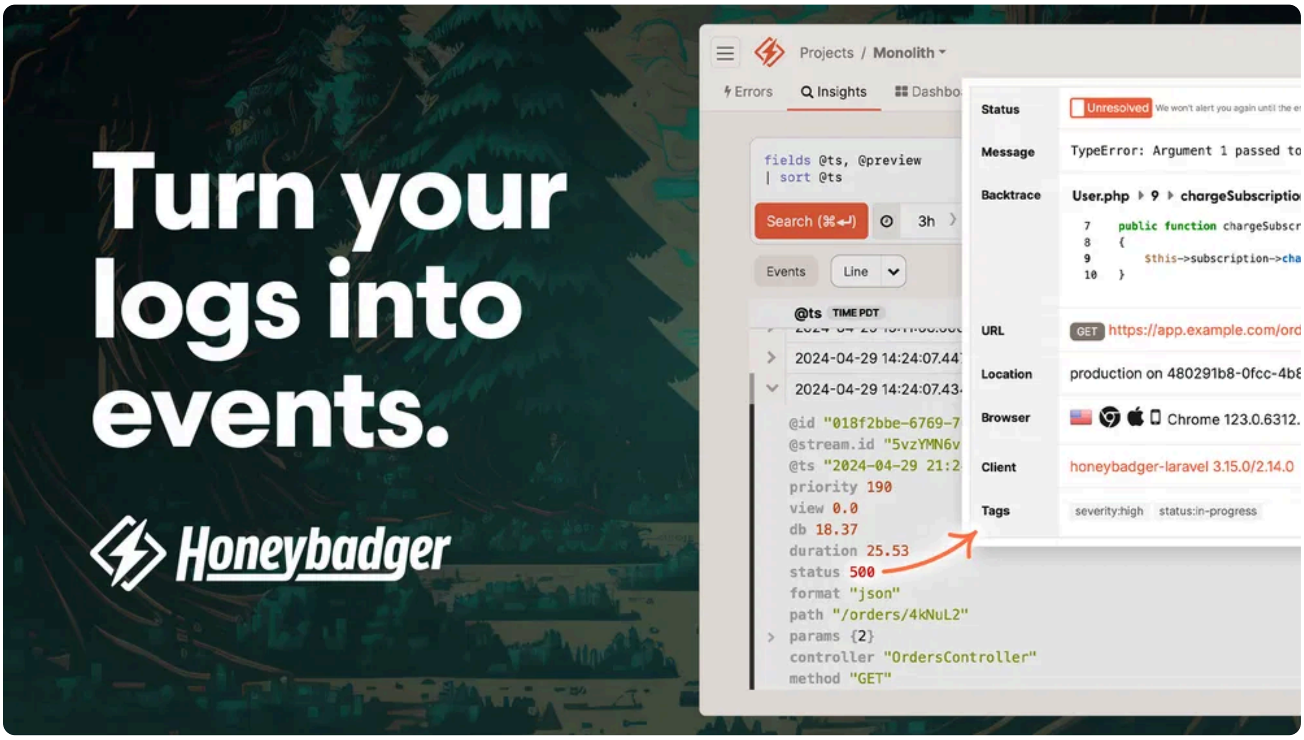
```
dir(object)

# Output:
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr_
'__sizeof__', '__str__', '__subclasshook__']
```

The above definition of the *Human* class is equivalent to the following code; here, we are explicitly inheriting the *object* base class. Although you can explicitly inherit the object base class, it's not required!

```
class Human(object):
    pass
```

object base class provides *__init__* and *__new__* methods that are used for creating and initializing objects of a class. We will discuss *__init__* and *__new__* in detail in the latter part of the tutorial.





"Splunk-like querying without having to sell my kidneys? nice"

That's a direct quote from someone who just saw Honeybadger Insights. It's a bit like Papertrail or DataDog—but with just the good parts and a reasonable price tag.

Best of all, **Insights logging is available on our free tier** as part of a comprehensive monitoring suite including error tracking, uptime monitoring, status pages, and more.

Start logging for FREE

Simple 5-minute setup – No credit card required

Objects and types in Python

Python is an object-oriented programming language. Everything in Python is an object or an instance. Classes, functions, and even simple data types, such as integer and float, are also objects of some class in Python. Each object has a class from which it is instantiated. To get the class or the type of object, Python provides us with the `type` function and `__class__` property defined on the object itself.

Let's understand the `type` function with the help of simple data types, such as `int` and `float`.

```
# A simple integer data type
a = 9

# The type of a is int (i.e., a is an object of class int)
type(a)    # Output: <class 'int'>

# The type of b is float (i.e., b is an object of the class float)
b = 9.0
type(b)   # Output: <class 'float'>
```

Unlike other languages, in Python, `9` is an object of class `int`, and it is referred by the variable `a`. Similarly, `9.0` is an object of class `float` and is referred by the variable `b`.

> `type` is used to find the type or class of an object. It accepts an object whose type we want to find out as the first argument and returns the type or class of that object.

We can also use the `__class__` property of the object to find the type or class of the object.

`__class__` is an attribute on the object that refers to the class from which the object was created.

```
a.__class__    # Output: <class 'int'>
b.__class__    # Output: <class 'float'>
```

After simple data types, let's now understand the *`type`* function and *`__class__`* attribute with the help of a user-defined class, *`Human`*. Consider the *`Human`* class defined below:

```
# Human class definition
class Human:
    pass

# Creating a Human object
human_obj = Human()
```

The above code creates an instance *`human_obj`* of the *`Human`* class. We can find out the class (or type of *`human_obj`*) from which *`human_obj`* was created using either the *`type`* function or the *`__class__`* property of the *`human_obj`* object.

```
# human_obj is of type Human
type(human_obj)     # Output: <class '__main__.Human'>

human_obj.__class__ # Output: <class '__main__.Human'>
```

The output of *`type(human_obj)`* and *`human_obj.__class__`* shows that *`human_obj`* is of type *`Human`* (i.e., *`human_obj`* has been created from the *`Human`* class).

As functions are also objects in Python, we can find their type or class using the *`type`* function or the *`__class__`* attribute.

```
# Check the type of the function
def simple_function():
    pass

type(simple_function)       # Output: <class 'function'>
simple_function.__class__   # Output: <class 'function'>
```

Thus, `simple_function` is an object of the class `function`.

> Classes from which objects are created are also objects in Python.

For example, the `Human` class (from which `human_obj` was created) is an object in itself. Yes, you heard it right! Even classes have a class from which they are created or instantiated.

Let's find out the type or class of the `Human` class.

```
class Human:
    pass

type(Human)        # Output: <class 'type'>
Human.__class__    # Output: <class 'type'>
```

Thus, the above code shows that the `Human` class and every other class in Python are objects of the class `type`. This `type` is a class and is different from the `type` function that returns the type of object. The `type` class, from which all the classes are created, is called the `Metaclass` in Python. Let's learn more about metaclass.

Metaclass in Python

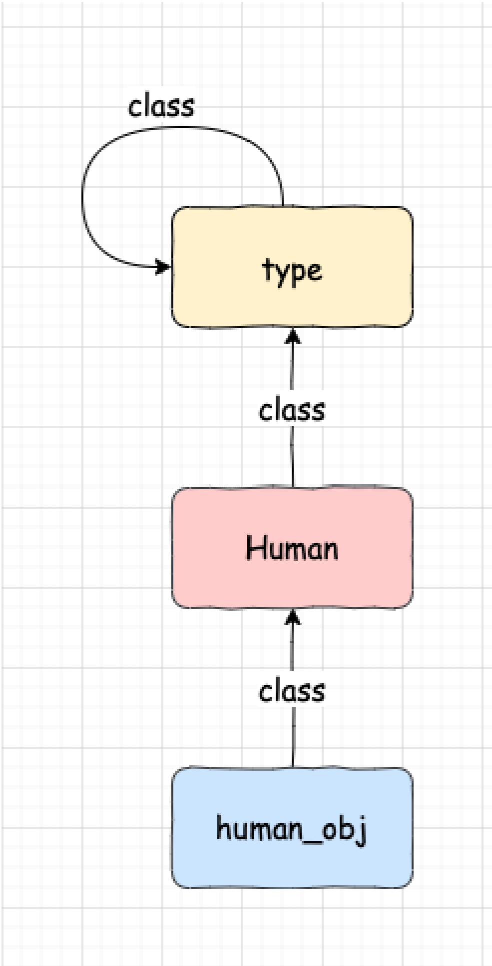
> Metaclass is a class from which classes are instantiated or metaclass is a class of a class.

Earlier in the article, we checked that variables `a` and `b` are objects of classes `int` and `float`, respectively. As `int` and `float` are classes, they should have a class or metaclass from which they are created.

```
type(int)    # Output: <class 'type'>
type(float)  # Output: <class 'type'>

# Even type of object class is - type
type(object) # Output: <class 'type'>
```

Thus, the *type* class is the metaclass of *int* and *float* classes. The *type* class is even the metaclass for the built-in *object* class, which is the base class for all the classes in Python. As *type* itself is a class, what is the metaclass of the *type* class? The *type* class is a metaclass of itself!

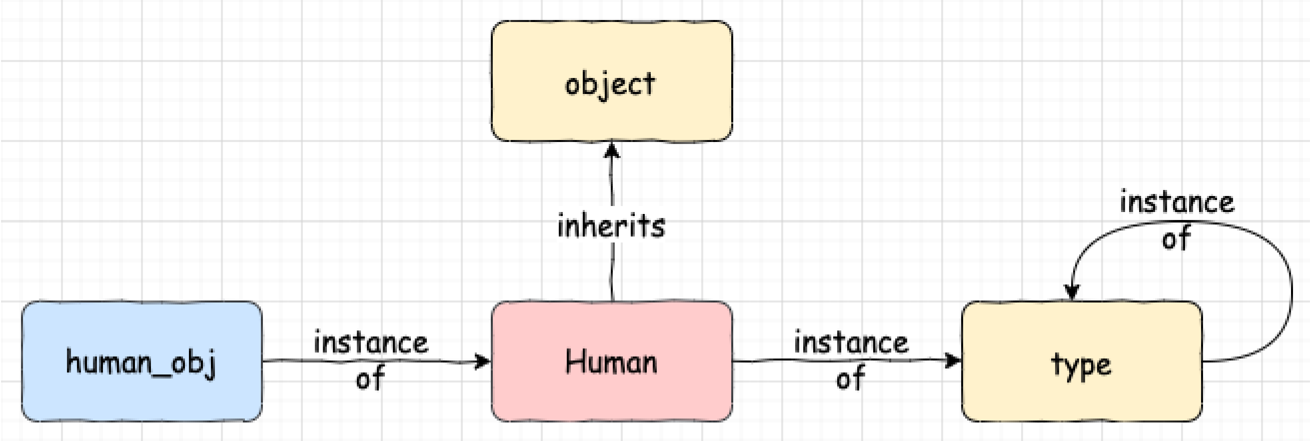
```
type(type) # Output: <class 'type'>
```



Metaclass is the least talked about topic and is not normally used very much in daily programming. I delve into this topic because metaclass plays an essential role in the object creation process that we will cover later in the article.

The two important concepts that we have covered so far are as follows:

1. All classes in Python are objects of the `type` class, and this `type` class is called `Metaclass`.
2. Each class in Python, by default, inherits from the `object` base class.



The object instantiation process in Python

With a basic understanding of the *Metaclass* and objects in Python, let's now understand the object creation and initialization process in Python. Consider the *Human* class, as defined below:

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

human_obj = Human("Virat", "Kohli")

isinstance(human_obj, Human) # Output: True

# As object is the base class for all the class hence
# isinstance(human_obj, object) is True
isinstance(human_obj, object) # Output: True
```

The output of the above code shows that *human_obj* is an instance of class *Human* with the *first_name* as *Virat* and the *last_name* as *Kohli*. If we look at the above code closely, it's natural to have some questions:

1. Per the definition of the `Human` class, we don't return anything from the `__init__` method; how does calling the `Human` class return the `human_obj`?
2. We know that the `__init__` method is used for initializing the object, but how does the `__init__` method get `self`?

In this section, we will discuss each of these questions in detail and answer them.

Object creation in Python is a two-step process. In the first step, Python creates the object, and in the second step, it initializes the object. Most of the time, we are only interested in the second step (i.e., the initialization step). Python uses the `__new__` method in the first step (i.e., object creation) and uses the `__init__` method in the second step (i.e., initialization).

If the class does not define these methods, they are inherited from the `object` base class. As the `Human` class does not define the `__new__` method, during the object instantiation process, the `__new__` method of the `object`'s class is called, while for initialization, the `__init__` method of the `Human` class is called. Next, we'll cover each of these methods in detail.

The `__new__` method

The `__new__` method is the first step in the object instantiation process. It is a static method on the `object` class and accepts `cls` or the class reference as the first parameter. The remaining arguments(`Virat` and `Kohli`) are passed while calling the class - `Human("Virat", "Kohli")`. The `__new__` method creates an instance of type `cls` (i.e., it allocates memory for the object by invoking the superclass' i.e. `object` class' `__new__` method using `super().__new__(cls)`). It then returns the instance of type `cls`.

Usually, it does not do any initialization, as that is the job of the `__init__` method. However, when you override the `__new__` method, you can also use it to initialize the object or modify it as required before returning it.

`__new__` method signature

```
# cls - is the mandatory argument. Object returned by the __new__ method is of type c
@staticmethod
def __new__(cls[,...]):
    pass
```

Override the `__new__` method

We can modify the object creation process by overriding the `__new__` method of the `object` class. Consider the example below:

```python
class Human:
    def __new__(cls, first_name=None):
        # cls = Human. cls is the class using which the object will be created.
        # Created object will be of type cls.
        # We must call the object class' __new__ to allocate memory
        obj = super().__new__(cls) # This is equivalent to object.__new__(cls)

        # Modify the object created
        if first_name:
            obj.name = first_name
        else:
            obj.name = "Virat"

        print(type(obj)) # Prints: <__main__.Human object at 0x103665668>
        # return the object
        return obj

# Create an object
# __init__ method of `object` class will be called.
virat = Human()

print(virat.name)  # Output: Virat

sachin = Human("Sachin")
print(sachin.name)  # Output: Sachin
```

In the above example, we have overridden the `__new__` method of the `object` class. It accepts the first arguments as `cls` - a class reference to the `Human` class.

> The `__new__` method is a special case in Python. Although it's a static method of the `object` class, on overriding it, we do not have to decorate it with the `staticmethod` decorator.

Inside the `__new__` method of the `Human` class, we are first calling the `__new__` method of the `object` class using `super().__new__(cls)`. The `object` class' `__new__` method creates and returns the instance of the class, which is passed as an argument to the `__new__` method. Here, as we are passing `cls` (i.e., the `Human` class reference); the `object`'s `__new__` method will return an instance of type `Human`.

> We must call the `object` class' `__new__` method inside the overridden `__new__` method to create the object and allocate memory to the object.

> The `__new__` method of the `Human` class modifies the `obj` returned from the `__new__` method of the `object` class and adds the `name` property to it. Thus, all objects created using the `Human` class will have a `name` property. Voila! We have modified the object instantiation process of the `Human` class.

Let's consider another example. In this example, we are creating a new class called `Animal` and overriding the `__new__` method. Here, when we are calling the `__new__` method of the `object` class from the `__new__` method of the `Animal` class, instead of passing the `Animal` class reference as an argument to the `__new__` method of the `object` class, we are passing the `Human` class reference. Hence, the object returned from the `__new__` method of the `object` class will be of type `Human` and not `Animal`. As a result, the object returned from calling the `Animal` class (i.e., `Animal()`) will be of type `Human`.

```python
class Animal:
    def __new__(cls):
        # Passing Human class reference instead of Animal class reference
        obj = super().__new__(Human) # This is equivalent to object.__new__(Human)

        print(f"Type of obj: {type(obj)}") # Prints: Type of obj: <class '__main__.Hu

        # return the object
        return obj

# Create an object
cat = Animal()
# Output:
# Type of obj: <class '__main__.Human'>

type(cat)   # Output: <class '__main__.Human'>
```

The `__init__` method

The `__init__` method is the second step of the object instantiation process in Python. It takes the first argument as an object or instance returned from the `__new__` method. The remaining arguments are the arguments passed while calling the class *(Human("Virat", "Kohli"))*. These arguments are used for initializing the object. The *__init__* method must not return anything. If you try to return anything using the *__init__* method, it will raise an exception, as shown below:

```
class Human:
    def __init__(self, first_name):
        self.first_name = first_name
        return self

human_obj = Human('Virat')
# Output: TypeError: __init__() should return None, not 'Human'
```

Consider a simple example to understand both the *__new__* and *__init__* method.

```
class Human:
    def __new__(cls, *args, **kwargs):
        # Here, the __new__ method of the object class must be called to create
        # and allocate the memory to the object
        print("Inside new method")
        print(f"args arguments {args}")
        print(f"kwargs arguments {kwargs}")

        # The code below calls the __new__ method of the object's class.
        # Object class' __new__ method allocates a memory
        # for the instance and returns that instance
        human_obj = super(Human, cls).__new__(cls)

        print(f"human_obj instance - {human_obj}")
        return human_obj

    # As we have overridden the __init__ method, the __init__ method of the object cl
    def __init__(self, first_name, last_name):
        print("Inside __init__ method")
        # self = human_obj returned from the __new__ method

        self.first_name = first_name
        self.last_name = last_name

        print(f"human_obj instance inside __init__ {self}: {self.first_name}, {self.l

human_obj = Human("Virat", "Kohli")

# Output
# Inside new method
```

```
# args arguments ('Virat', 'Kohli')
# kwargs arguments {}
# human_obj instance - <__main__.Human object at 0x103376630>
# Inside __init__ method
# human_obj instance inside __init__ <__main__.Human object at 0x103376630>: Virat, K
```

In the above code, we have overridden both the `__new__` and `__init__` method of the `object`'s class. `__new__` creates the object (`human_obj`) of type `Human` class and returns it. Once the `__new__` method is complete, Python calls the `__init__` method with the `human_obj` object as the first argument. The `__init__` method initializes the `human_obj` with `first_name` as `Virat` and `last_name` as `Kohli`. As object creation is the first step, and initialization is the second step, the `__new__` method will always be called before the `__init__` method

Both `__init__` and `__new__` are called magic methods in Python. Magic methods have names that begin and end with __ (double underscores or "dunder"). Magic methods are called implicitly by the Python; you do not have to call them explicitly. For example, both the `__new__` and `__init__` method are called implicitly by Python. Let's cover one more magic method, `__call__`.

The `__call__` method

The `__call__` method is a magic method in Python that is used to make the objects callable. Callable objects are objects that can be called. For example, `functions` are callable objects, as they can be called using the round parenthesis.

Consider an example to better understand callable objects:

```
def print_function():
    print("I am a callable object")

# print_function is callable as it can be called using round parentheses
print_function()

# Output
# I am a callable object
```

Let's try to call an *integer* object. As *integer* objects are not callable, calling them will raise an exception.

```
a = 10

# As the integer object is not callable, calling `a` using round parentheses will rai
a()   # Output: TypeError: 'int' object is not callable
```

callable()

The *callable* function is used to determine whether an object is callable. The *callable* function takes the object reference as an argument and returns *True* if the object appears to be callable or *False* if the object is not callable. If the *callable* function returns *True*, the object might not be callable; however, if it returns *False*, then the object is certainly not callable.

```
# Functions are callable
callable(print_function)
# Output: True

# Interger object is not callable
callable(a)
# Output: False
```

Let's determine whether the classes in Python are callable. Here, we will determine whether the *Human* class defined earlier is callable.

```
callable(Human)
# Output: True
```

Yes, classes in Python are callable, and they should be! Don't you think so? When we call the class, it returns the instance of that class. Let's find out whether the objects created from the class are callable.

```
human_obj = Human("Virat", "Kohli")

callable(human_obj)   # Output: False

# Let's try calling the human_obj
human_obj()
```

```
# As human_obj is not callable it raises an exception
# Output: TypeError: 'Human' object is not callable
```

So, *human_obj* is not callable though the class of *human_obj* (i.e., the *Human* class is callable).

To make any object in Python callable, Python provides the *__call__* method that needs to be implemented by the object's class. For example, to make *human_obj* object callable, the *Human* class has to implement the *__call__* method. Once the *Human* class implements the *__call__* method, all the objects of the *Human* class can be invoked like functions (i.e., using round parentheses).

```
class Human:
    def __init__(self, first_name, last_name):
        print("I am inside __init__ method")
        self.first_name = first_name
        self.last_name = last_name

    def __call__(cls):
        print("I am inside __call__ method")

human_obj = Human("Virat", "Kohli")
# Output: I am inside __init__ method

# Both human_obj() and human_obj.__call__() are equaivalent
human_obj()
# Output: I am inside __call__ method

human_obj.__call__()
# Output: I am inside __call__ method

callable(human_obj)
# Output: True
```

The above code output shows that after implementing the *__call__* method on the *Human* class,*human_obj* becomes a callable object. We can call the *human_obj* using round parentheses (i.e., *human_obj()*). When we use *human_obj()*, in the background, Python calls the *__call__* method of the *Human* class. So, instead of calling *human_obj* as *human_obj()*, we can directly invoke the *__call__* method on *human_obj* (i.e., *human_obj.__call__()*). Both *human_obj()* and *human_obj.__call__()* are equivalent, and they are the same thing.

> For all objects that are callable, their classes must implement the `__call__` method.

We know that functions are a callable object, so its class (i.e., *`function`*) must implement the `__call__` method. Let's invoke the `__call__` method on the *`print_function`* defined earlier.

```
print_function.__call__()   # Output: I am a callable object
```

> In Python, *`class`* is also a callable object; therefore, it is a *`class`*'s class (metaclass) (i.e., the *`type`* class must have a *`call`* method defined on it). Hence, when we call *`Human()`*, in the background, Python calls the *`call`* method of the *`type`* class.

Roughly, the `__call__` method on the *`types`* class looks something like shown below. This is just for explanation purposes; we will cover the actual definition of the `__call__` method later in the tutorial.

```
class type:
    def __call__():
        # Called when class is called i.e. Human()
        print("type's call method")
```

With an understanding of `__call__` method and how calling the class calls the `__call__` method of the *`type`* class, let's find out the answer to the following questions regarding the object initialization process:

1. Who calls the `__new__` and `__init__` method?
2. Who passes the `self` object to the `__init__` method?
3. As the `__init__` method is called after the `__new__` method, and the `__init__` method does not return anything, how does calling the class return the object (i.e., how does calling the `Human` class return the `human_obj` object)?

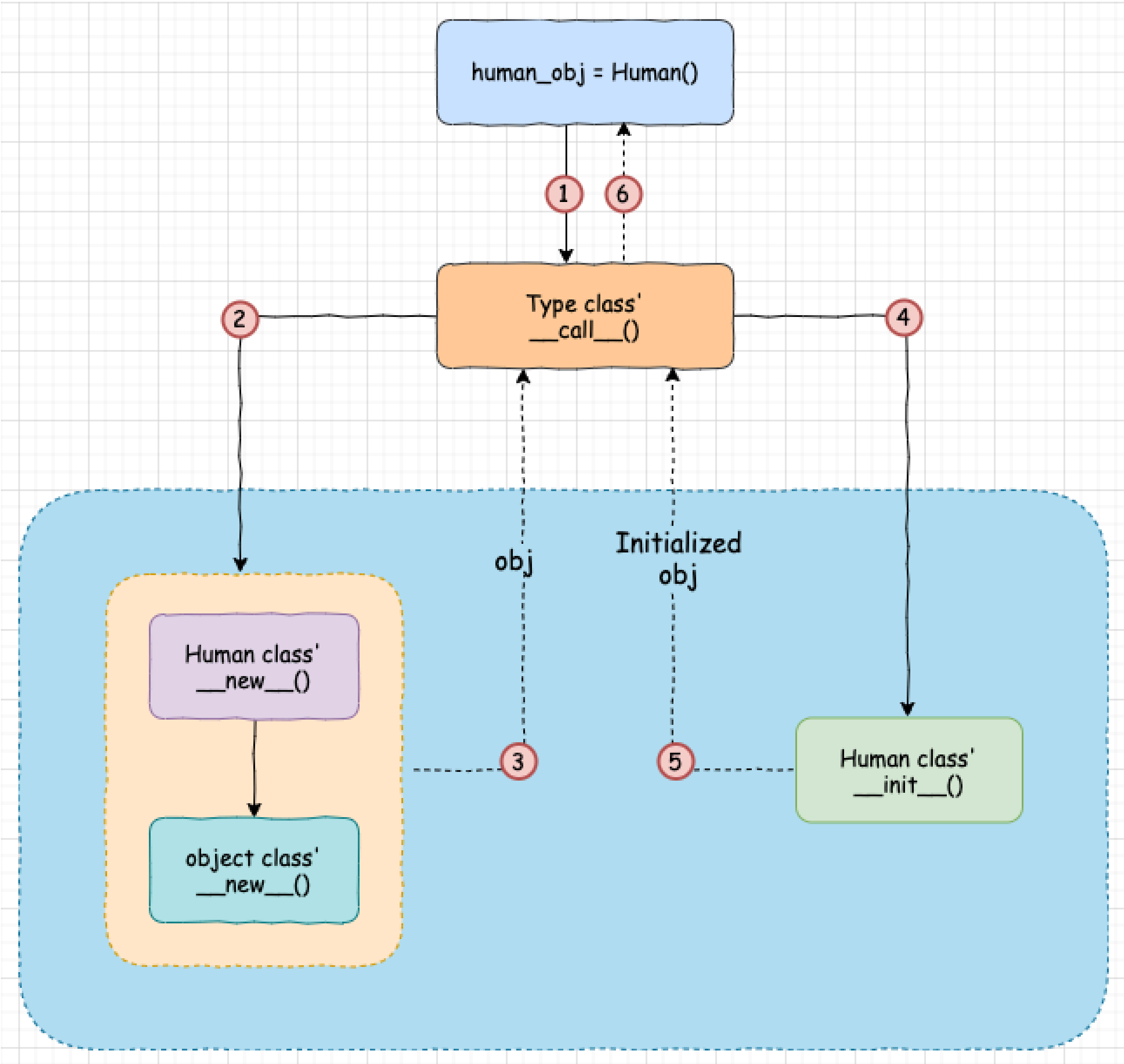
Consider an example of instantiating an object in Python.

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


human_obj = Human("Virat", "Kohli")
```

We know that when we call the class (i.e., *Human("Virat", "Kohli")*), the *__call__* method of the *type* class is called. However, what is the definition of the *types* class' *__call__* method? As we are talking about CPython, the *type* class' *__call__* method [definition](#) is defined in C language. If we convert it into Python and simplify it, it will look somewhat like this:

```
# type's __call__ method which gets called when Human class is called i.e. Human()
def __call__(cls, *args, **kwargs):
    # cls = Human class
    # args = ["Virat", "Kohli"]
    # Calling __new__ method of the Human class, as __new__ method is not defined
    # on Human, __new__ method of the object class is called
    human_obj = cls.__new__(*args, **kwargs)

    # After __new__ method returns the object, __init__ method will only be called if
    # 1. human_obj is not None
    # 2. human_obj is an instance of class Human
    # 3. __init__ method is defined on the Human class
    if human_obj is not None and isinstance(human_obj, cls) and hasattr(human_obj, '_
        # As __init__ is called on human_obj, self will be equal to human_obj in __ir
        human_obj.init(*args, **kwargs)

    return human_obj
```

Let's understand the above code; when we do *Human("Virat", "Kohli")*, in the background, Python will call the *type* class' *__call__* method, which is defined like the above code snippet. As shown above, the *type* class' *__call__* method accepts *Human* class as the first argument (*cls* is *Human* class), and the remaining arguments are passed while calling the *Human* class. The *type* class' *__call__* method will first call the *__new__* method defined on the *Human* class, if any; otherwise, the *__new__* method of the *Human* class' parent class (i.e. the *object*'s *__new__* method) is called. The *__new__* method will return the *human_obj*. Now, the *__call__* method of the *type* class will call the *__init__* method defined on the *`Human`* class with *`human_obj`* as the first argument. *`__init__`* will initialize the *`human_obj`* with the passed arguments, and finally, the *`__call__`* method will return the *`human_obj`*.



So, following steps are followed while creating and initializing an object in Python:

1. Call the `Human` class - `Human()`; this internally calls the `__call__` method of the `type` class (i.e., `type.__call__(Human, "Virat", "Kohli")`).
2. `type.__call__` will first call the `__new__` method defined on the `Human` class. If the `__new__` method is not defined on the `Human` class, the `__new__` method of the `object` class will be called.
3. The `__new__` method will the return the object of type `Human` i.e. `human_obj`

4. Now, `type.__call__` will call the `__init__` method defined on the `Human` class with `human_obj` as the first argument. This `human_obj` will be `self` in the `__init__` method.
5. The `__init__` method will initialize the `human_obj` with the `first_name` as `Virat` and the`last_name` as `Kohli`. The `__init__` method will not return anything.
6. In the end, `type.__call__` will return the `human_obj` object.

As per the *`type.__call__`* definition, whenever we create a new object, the *`__new__`* method will always be called, but calling the *`__init__`* method depends on the output of the *`__new__`* method. The *`__init__`* method will be called only if the *`__new__`* method returns an object of type *`Human`* class or a subclass of the *`Human`* class.

Let's understand some of the cases.

Case 1: If the returned object from the *`__new__`* method is of type *`Human`* (i.e., the class of the *`__init__`* method), the *`__init__`* method will be called.

```
class Human:
    def __new__(cls, *args, **kwargs):
        print(f"Creating the object with cls: {cls} and args: {args}")
        obj = super().__new__(cls)
        print(f"Object created with obj: {obj} and type: {type(obj)}")
        return obj

    def __init__(self, first_name, last_name):
        print(f"Started: __init__ method of Human class with self: {self}")
        self.first_name = first_name
        self.last_name = last_name
        print(f"Ended: __init__ method of Human class")

human_obj = Human("Virat", "Kohli")
```

Output:

```
# Creating the object with cls: <class '__main__.Human'> and args: ('Virat', 'Kohli')
# Object created with obj: <__main__.Human object at 0x102f6a4e0> and type: <class '_
# Started: __init__ method of Human class with self: <__main__.Human object at 0x102f
# Ended: __init__ method of Human class with self: <__main__.Human object at 0x102f6a
```

Case 2: If the *__new__* method does not return anything, then *__init__* will not be called.

```
class Human:
    def __new__(cls, *args, **kwargs):
        print(f"Creating the object with cls: {cls} and args: {args}")
        obj = super().__new__(cls)
        print(f"Object created with obj: {obj} and type: {type(obj)}")
        print("Not returning object from __new__ method, hence __init__ method will r

    def __init__(self, first_name, last_name):
        print(f"Started: __init__ method of Human class with self: {self}")
        self.first_name = first_name
        self.last_name = last_name
        print(f"Ended: __init__ method of Human class")

human_obj = Human("Virat", "Kohli")
```

Output:

```
# Creating the object with cls: <class '__main__.Human'> and args: ('Virat', 'Kohli')
# Object created with obj: <__main__.Human object at 0x102f6a5c0> and type: <class '_
# Not returning object from __new__ method, hence __init__ method will not be called
```

In the above code, the *__new__* method of the *Human* class is called; hence, *obj* of type *Human* is created (i.e., memory is assigned for *obj*). However, as the *__new__* method did not return *human_obj*, the *__init__* method will not be called. Also, *human_obj* will not have the reference for the created object, as it was not returned from the *__new__* method.

```
print(human_obj). # Output: None
```

Case 3: The *__new__* method returns an integer object.

```
class Human:
    def __new__(cls, *args, **kwargs):
        print(f"Creating the object with cls: {cls} and args: {args}")
        obj = super().__new__(cls)
        print(f"Object created with obj: {obj} and type: {type(obj)}")
        print("Not returning object from __new__ method, hence __init__ method will r
        return 10
```

```
    def __init__(self, first_name, last_name):
        print(f"Started: __init__ method of Human class with self: {self}")
        self.first_name = first_name
        self.last_name = last_name
        print(f"Ended: __init__ method of Human class")

human_obj = Human("Virat", "Kohli")
```

In the above code, the `__new__` method of the `Human` class is called; hence, `obj` of type `Human` is created (i.e., memory is assigned for `obj`). However, the `__new__` method did not return `human_obj` but an integer with value `10`, which is not of the `Human` type; hence, the `__init__` method will not be called. Also, `human_obj` will not have the reference for the created object, but it will refer to an integer value of `10`.

```
print(human_obj)
# Output: 10
```

In scenarios where the `__new__` method does not return the instance of the class and we want to initialize the object, we have to call the `__init__` method, explicity, inside the `__new__` method, as shown below:

```
class Human:
    def __new__(cls, *args, **kwargs):
        print(f"Creating the object with cls: {cls} and args: {args}")
        obj = super().__new__(cls)
        print(f"Object created with obj: {obj} and type: {type(obj)}")
        print("Not returning object from __new__ method, hence __init__ method will r
        obj.__init__(*args, **kwargs)
        return 10

    def __init__(self, first_name, last_name):
        print(f"Started: __init__ method of Human class with self: {self}")
        self.first_name = first_name
        self.last_name = last_name
        print(f"Ended: __init__ method of Human class")

human_obj = Human("Virat", "Kohli")
```

Output:

```
# Creating the object with cls: <class '__main__.Human'> and args: ('Virat', 'Kohli')
# Object created with obj: <__main__.Human object at 0x102f6a860> and type: <class '_
# Not returning object from __new__ method, hence __init__ method will not be called
# Started: __init__ method of Human class with self: <__main__.Human object at 0x102f
# Ended: __init__ method of Human class
```

In the above case, the `__new__` method returned an integer object; hence the `human_obj` value will be 10.

```
print(human_obj)
# Output: 10
```

Conclusion

In this article, we explored the `__new__`, `__init__`, and `__call__` magic methods and discussed Metaclass in Python. In doing so, we have a better understanding of the object creation and initialization processes in Python.

References

1. https://eli.thegreenplace.net/2012/04/16/python-object-creation-sequence
2. https://realpython.com/python-metaclasses/#old-style-vs-new-style-classes
3. https://docs.python.org/3/reference/datamodel.html#special-method-names

Get the Honeybadger newsletter

Each month we share news, best practices, and stories from the DevOps & monitoring community—exclusively for developers like you.

Your first name

Your email address

Sign up

☑ Include latest Python articles

Rupesh Mishra

Rupesh Mishra is a backend developer, freelance blogger, and tutor. He writes about Python, Docker, Kafka, Kubernetes, and MongoDB. When he is not coding he enjoys watching anime and movies.

X/Twitter

More articles by Rupesh Mishra

More articles

Nov 01, 2024 Your guide to reducing Python memory usage
Mar 19, 2024 Building command-line applications in Python
Feb 20, 2024 How to dockerize a Django, Preact, and PostgreSQL Application
Sep 28, 2023 Options for passwordless authentication in Django apps
Aug 31, 2023 A guide to exception handling in Python
Aug 10, 2023 Beginners guide to software testing in Python
Jul 20, 2023 Tips for upgrading Python/Django versions in existing apps
Jun 15, 2023 A comprehensive guide on how to migrate from Python to Go
May 25, 2023 Machine-learning life-cycle management using MLflow
May 04, 2023 Code coverage vs. test coverage in Python

Product
Error Tracking
Uptime Monitoring
Status Pages
Logging & Observability
Cron & Heartbeat
Monitoring
Integrations
Plans & pricing

Stacks
Rails
Laravel
Django
Phoenix
JavaScript
Ruby
Node
Python

GDPR
Security

PHP
Elixir
Crystal
Go
Cocoa

Company
Meet the 'Badgers
Job openings
Brand assets
Terms of use
Privacy statement
Contact us

Resources
Developer docs
Developer blog
Newsletter
Exceptional Creatures
FounderQuest

Switching to Honeybadger
Alternative to Sentry
Alternative to Rollbar
Alternative to BugSnag
Alternative to Airbrake
See all comparisons