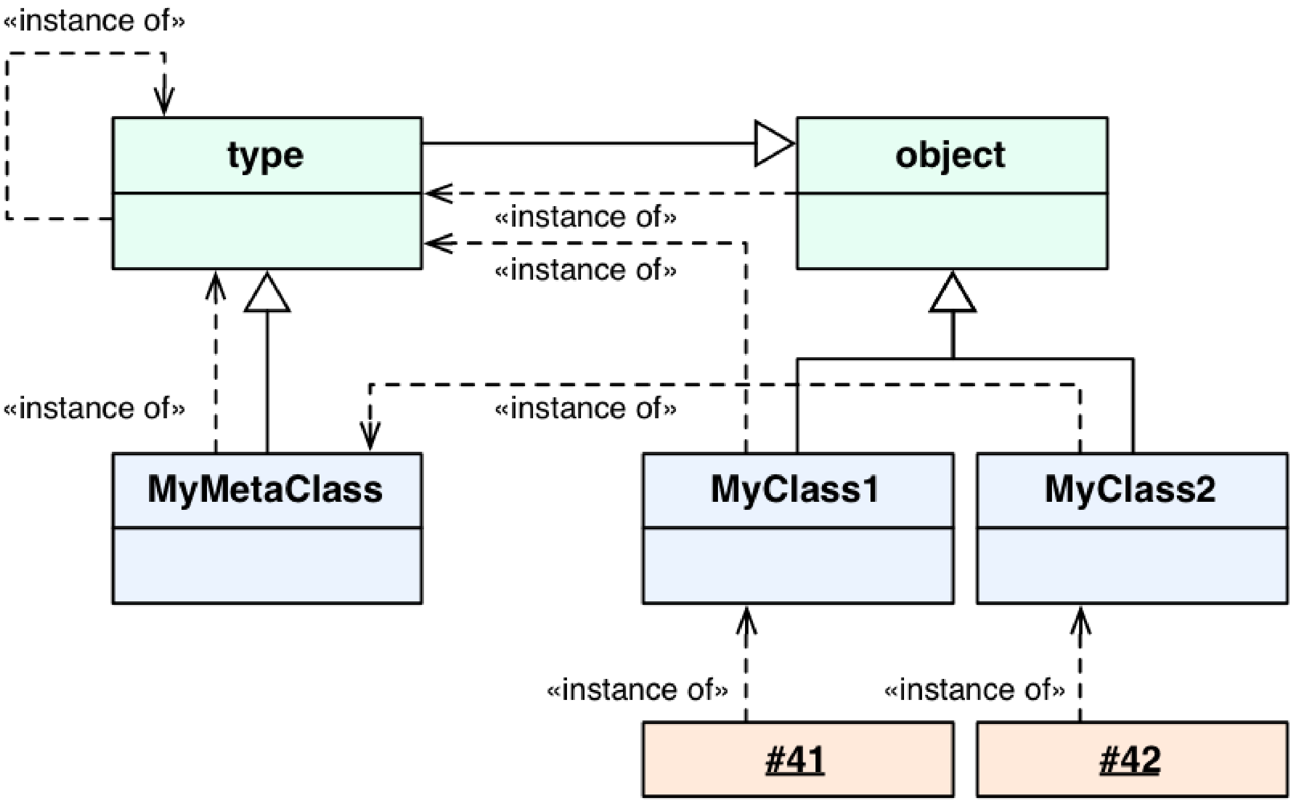
Andrew

Follow

13 Followers

About



Mutable and Immutable Objects in Python

Andrew May 8, 2018 · 7 min read

In Python, everything is an object. In order to understand what this means we will need to understand how Python is set up. In Python's documentation:

> Objects *are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects.*

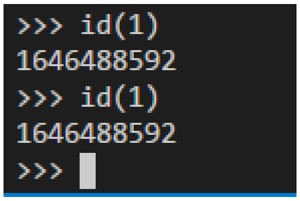
associated with it. This type determines
An ID is an integer or long integer that u
number is the memory address for that
mutable and immutable objects? These
Python’s memory allocation works with
they resolved after different operations.

So What Are Types, Classes and Instances?

As mentioned before, all objects in Python have a type. This type is like the parent of the object related to it. For example, the number 1 is an object of the type int. Classes are something that was an issue in older Python versions. Old style classes are generally user-defined while types are built ins. In current versions of Python, this has been changed and there is now no real difference between the two. If anything, class is now a way for users to make user defined types. Instances are pretty much objects. That is to say while 1 is an int object; 1 is an instance of type int.

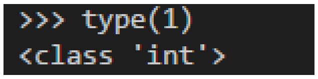
The id() and type() built-ins

As was briefly mentioned, the id() builtin in CPython gives back an integer that ties to the memory address of a particular object during its lifetime. For example, if I were to type `id(1),` I will be given a unique integer tied to the object 1. I have run the command twice in the example below to show that the id is indeed unique and 1 is an instance, or object, of type int.



```
>>> id(1)
1646488592
>>> id(1)
1646488592
>>>
```

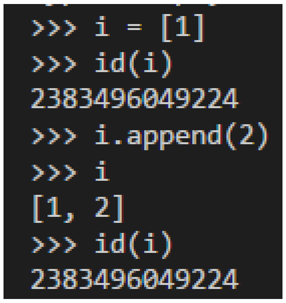
The `type()` built-in returns what the type of an object is. It should be noted that since everything is an object even types have a type, which is type.



```
>>> type(1)
<class 'int'>
```

Mutable Objects

Based on namesake alone, it can be infer
being changed while keeping its id. In Py
dictionaries, sets, bytearray, and etc. A q
mutable is through the use of the `id()` builtin.

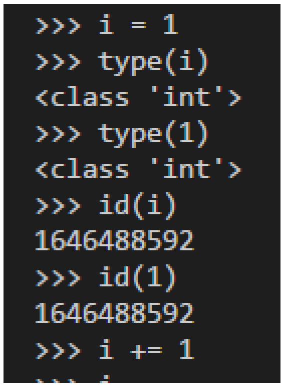


```
>>> i = [1]
>>> id(i)
2383496049224
>>> i.append(2)
>>> i
[1, 2]
>>> id(i)
2383496049224
```

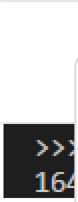
In the example above, I have made an object of type list `i` with the value of 1 in it. After creating the list, I have appended a 2 at the end of the list. The append built in mutates the list by increasing its length by one and adds in its argument at the end of the list. **I have taken the id of object i before and after the change and as you can see, they are the same.** This means that the object i has been changed but kept its unique id and subsequently its memory address.

Immutable objects

Immutable objects are the direct opposite of mutable objects as **they do not allow changes after its creation**. Such object types are: integers, float, complex, strings, tuple, frozensets, and bytes.



```
>>> i = 1
>>> type(i)
<class 'int'>
>>> type(1)
<class 'int'>
>>> id(i)
1646488592
>>> id(1)
1646488592
>>> i += 1
```



```
>>:
164
```

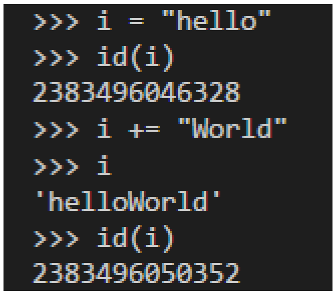
In this example I have made an object i

when we add 1 to i, we will have 2 and a

the numbers themselves and we can see that the id of i is a direct copy of the ids of the integers. This is expected behavior because within python, there is an array of integer objects from -5 to 256 preallocated for us. Whenever we reference to a number within this range, we are actually referring to an existing object in the memory instead of creating a new object.

Why Should I Care If It's Immutable or Mutable?

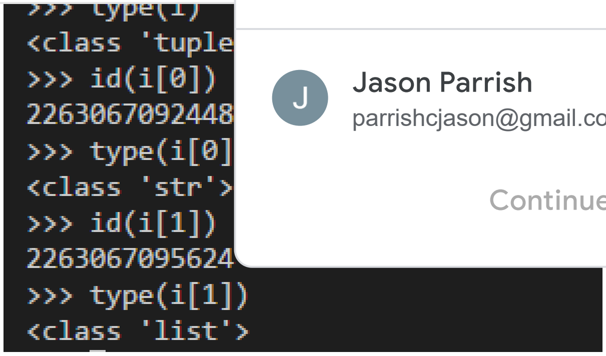
Since immutable objects are impervious to changes, what happens when we try to concatenate two immutable objects like two strings?



```
>>> i = "hello"
>>> id(i)
2383496046328
>>> i += "World"
>>> i
'helloWorld'
>>> id(i)
2383496050352
```

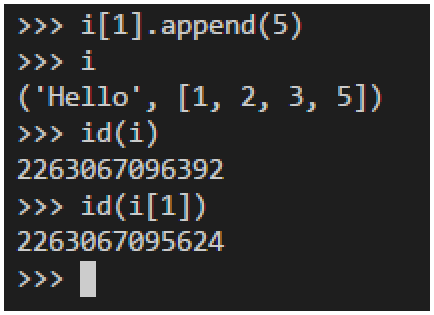
As we can see, the id for object i was changed as we concatenated the string `"World"` to it. In this operation, the concatenation of i and the string created a new object i with type string that contains the value of both the old object i and the string "World". This means that "changing" an immutable type is an arduous process as a new copy is needed. This can be a very expensive process for extremely long strings or dictionaries. On the flip side, mutable objects are easier to mutate.

However, the inflexibility of immutable types can come in handy. Since the actual object cannot be changed unlike mutable objects, we can be certain that the object will stay the same unless an operation is used. There is a loophole to this immutable rule though. What happens when you add a mutable object within an immutable object like a tuple?



```
>>> type(i)
<class 'tuple
>>> id(i[0])
2263067092448
>>> type(i[0]
<class 'str'>
>>> id(i[1])
2263067095624
>>> type(i[1])
<class 'list'>
```

Here, I have created an object i with the class tuple which holds a reference to a string, which is immutable, and a list, which is mutable.

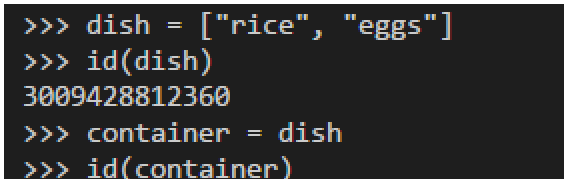


```
>>> i[1].append(5)
>>> i
('Hello', [1, 2, 3, 5])
>>> id(i)
2263067096392
>>> id(i[1])
2263067095624
>>>
```

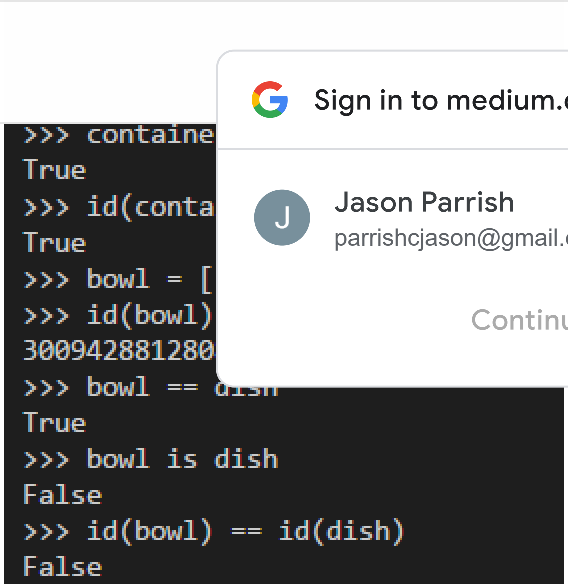
Since the list is mutable, I can use list built ins to modify the list itself. As we can see, none of the ids have changed meaning the mutable object list within the immutable tuple is the same as before. So what is immutable is the content of the tuple itself meaning the references to the string and the list. So this means that immutable objects within an immutable object cannot be changed as the only way to “change”an immutable object is to make a new object with the updated values.

A Small Intro to == and is

Within Python there are two operators `==` and `is`. The `==` operator checks for values while the `is` operator checks for identities(id). This means that `x is y` is similar to `id(x) == id(y)`.



```
>>> dish = ["rice", "eggs"]
>>> id(dish)
3009428812360
>>> container = dish
>>> id(container)
```

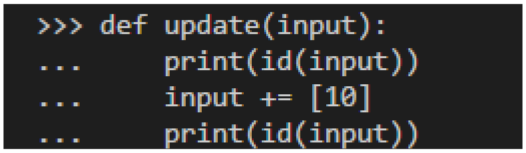


```
>>> containe
True
>>> id(conta
True
>>> bowl = [
>>> id(bowl)
300942881280
>>> bowl == dish
True
>>> bowl is dish
False
>>> id(bowl) == id(dish)
False
```

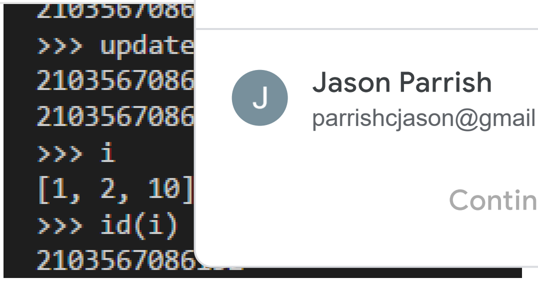
In the first line, I have created a list called dish that contains two strings, spam and eggs. I have created an alias of dish which is container so now container and dish point to the same list. When I do a value comparison check with `==` , I get back true because they both contain the same two strings. I also do an identity comparison using `is` and I get back True as well. Next, I have created a new list called bowl with the same two strings in the list. I do a value comparison check with `==` between bowl and dish and I get back true. However, when I do an identity check with `is` between bowl and dish, I get False.

How Are Arguments Passed to Functions?

Learning the difference between mutable and immutable objects is important to understand how they are treated when passed to functions(parameter passing) and how memory allocation works in Python. In programming, there are two main ways of passing arguments to functions: call by value and call by reference. Call by value means that only the value of the argument and not the argument itself was sent to the function. This means that the original argument hasn't changed after the program resolves. Call by reference is when the program gets the memory address of the argument. This means that any changes made within the program can be seen after the program resolves. In the Python documentation, arguments are passed by assignment. This means that arguments are passed to functions as references to the object(argument).

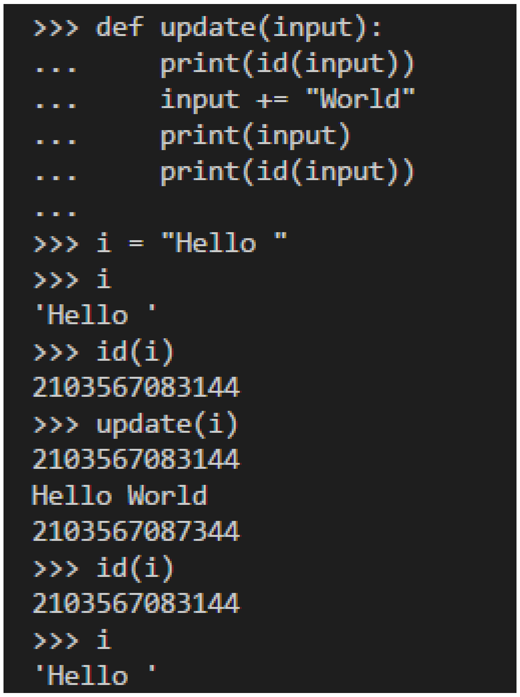


```
>>> def update(input):
...     print(id(input))
...     input += [10]
...     print(id(input))
```



```
2103567086
>>> update
2103567086
2103567086
>>> i
[1, 2, 10]
>>> id(i)
2103567086
```

In the example above, I have made a simple program that takes in an input and adds a list [10] to the input. I have passed in i which is the list [1,2] and got back an updated list with the 10 in it. As you have noticed, all of the addresses are the same whether inside or outside the function. Since the object i was a mutable object, Python has resolved the parameter passing of i similar to call by reference.



```
>>> def update(input):
...     print(id(input))
...     input += "World"
...     print(input)
...     print(id(input))
...
>>> i = "Hello "
>>> i
'Hello '
>>> id(i)
2103567083144
>>> update(i)
2103567083144
Hello World
2103567087344
>>> id(i)
2103567083144
>>> i
'Hello '
```

In this example, I have made a program that will add a string”World” to the input, which will also be a string. Based on the ids printed, we can see that when an immutable value is passed into the function, Python behaves like a call by reference. But since we are “changing” an immutable object, Python’s behavior “switches” to call by value and creates a new string object to hold the newly created object. Of course, the original input was never changed as the new object created within the program was never saved anywhere and was discarded.

Programming Python Oop