Delete in Python

Instantiating classes

To instantiate a class, simply call the class as if it were a function, passing the arguments that the **__init__()** method requires. The return value will be the newly created object. In Python, there is no explicit *new* operator like there is in C++ or Java. So, we simply call a class as if it were a function to create a new instance of the class:

```
s = Student(args)
```

We are creating an instance of the **Student** class and assigning the newly created instance to the variable **s**. We are passing one parameter, **args**, which will end up as the argument in Student's **__init__()** method.

s is now an instance of the **Student** class. Every class instance has a built-in attribute, **__class__**, which is the object's class. Java programmers may be familiar with the Class class, which contains methods like **getName()** and **getSuperclass()** to get metadata information about an object. In Python, this kind of metadata is available through attributes, but the idea is the same.

We can access the instance's **docstring** just as with a function or a module. All instances of a class share the same docstring.

We can use the **Student** class defined above as following:

```
studentA = Student("Jack")
studentB = Student("Judy", 10005)
```

Unlike C++, the attributes of Python object are public, we can access them using the dot(.) operator:

```
>>>studentA.name
'Jack'
>>>studentB.id
10005
```

We can also assign a new value to the attribute:

```
>>> studentB.id = 80001
>>> studentB.id
80001
```

How about the object destruction?

Python has automatic garbage collection. Actually, when an object is about to be garbage-collected, its **__del()__** method is called, with **self** as its only argument. But we rarely use this method.

`object.__del__(self)`

Called when the instance is about to be destroyed. This is also called a destructor. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible (though not recommended!) for the `__del__()` method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. It is not guaranteed that `__del__()` methods are called for objects that still exist when the interpreter exits.

Note

`del x` doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero. Some common situations that may prevent the reference count of an object from going to zero include: circular references between objects (e.g., a doubly-linked list or a tree data structure with parent and child pointers); a reference to the object on the stack frame of a function that caught an exception (the traceback stored in `sys.exc_info()[2]` keeps the stack frame alive); or a reference to the object on the stack frame that raised an unhandled exception in interactive mode (the traceback stored in `sys.last_traceback` keeps the stack frame alive). The first situation can only be remedied by explicitly breaking the cycles; the second can be resolved by freeing the reference to the traceback object when it is no longer useful, and the third can be resolved by storing `None` in `sys.last_traceback`. Circular references which are garbage are detected and cleaned up when the cyclic garbage collector is enabled (it's on by default). Refer to the documentation for the `gc` module for more information about this topic.

`__del__` is a **destructor**. It is called when an object is **garbage collected** which happens after all references to the object have been deleted.

In a **simple case** this could be right after you say `del x` or, if `x` is a local variable, after the function ends. In particular, unless there are circular references, CPython (the standard Python implementation) will garbage collect immediately.

However, this is the **implementation detail** of CPython. The only **required** property of Python garbage collection is that it happens *after* all references have been deleted, so this might not necessary happen *right after* and **might not happen at all**.

The `__del__` method, it will be called when the object is garbage collected. Note that it isn't necessarily guaranteed to be called though. The following code by itself won't necessarily do it:

```
del obj
```

The reason being that `del` just decrements the reference count by one. If something else has a reference to the object, `__del__` won't get called.

There are a few caveats to using `__del__` though. Generally, they usually just aren't very useful…

See the python documentation on `__del__` methods.

One other thing to note: `__del__` methods can inhibit garbage collection if overused. In particular, a circular reference that has more than one object with a `__del__` method won't get garbage collected. This is because the garbage collector doesn't know which one to call first. See the documentation on the gc module for more info.

Some References:

https://docs.python.org/3/reference/datamodel.html#object.__del__

Stackoverflow