From Perl to Python

Fernando Pineda 140.636 version 0.3

1. Introduction

If Python is ever to make any sense there are two things you **MUST** understand about the language:

1. Everything is an object
2. Unlike many other languages (Perl, C, R, etc.) *variables are not containers for values.* Instead, they are more like *references* in Perl, or *pointers* in C.

Three functions will help us wrap our minds around this:

- `type()` Every object has a type/class (in Python2/Python3 respectively)
- `id()` Every object has a *usually* unique identifier.
- `dir()` Every object has a list of names defined in it's scope.

1. An object has a type (class)

To follow along, start the Python3 interpreter.

Classes are also called *types*. In Python2 `type` was actually different from `class`. The difference is unimportant (for us). The difference between `class` and `type` disappeared in python3

Again: Literally everything is an object. Let this sink-in by trying the following trivial statement:

```
>>> 0
0
>>> type(0)
<class 'int'>
```

When you typed `0` Python created an object of class `int` and then saved the value `51` as a attribute of that object. We'll elaborate on this later.

There are three built-in numerical classes in Python: `int`, `float` and `complex`.

I really mean **everything** is an object. Try this:

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

This first statement shows that the built-in object `int` is an object of class 'type' and that the class 'type' is itself an object of class `type`. (Note: in Python2, a class is an instance of the class "type", but in Python3 `type` means `classes`.)Refer to the documentation for a comprehensive explanation of python built-in types:

https://docs.python.org/3/library/stdtypes.html#classes-and-class-instances

2. An object has a unique identifier

An object is an instance of a class. Each instance has a unique identifier (At least as long as it exists). Let's revisit our trivial statement:

```
>>> 0
0
>>>
```

Here is what happened

1. Python allocated memory for an object of class `int`
2. Python saved the integer value 0 as an attribute of the object

3. Python printed the value of the object (hence 0)
4. Python was finished with the object and marked it for deletion
5. Python displayed a prompt and waited for you to type something
6. Python will delete (free the memory of) the unused `int` object when it is good and ready.

Let's explore this in more detail. First let's create an object of class `str` and check it's class with `type()`:

```
>>> type("foo")
<class 'str'>
```

Great. Now let's get it's identifier with `id()`:

```
>>> id("foo")
5853344
```

This is the unique id of the int object with value 51.

Now try repeatedly entering `id("foo")`. You will either see something like this:

```
  >>> id("foo")     # create a string object and print its unique id
  4449499456
  >>> id("foo")     # create another string object and print its unique id
  4449499416
  >>> id("foo")     # create yet another string object and print its unique id
  4449499576
```

or something like this:

```
>>> id("foo")     # create a string object and print its unique id
4449499456
>>> id("foo")     # reuses the same object because it hasn't deleted the first one
4449499456
>>> id("foo")     # reuses the same object because it hasn't deleted the first one
4449499456
```

In the CPython variant of Python, this is simply the memory address of the object. Moreover, according to the documentation, this integer is guaranteed to be unique and constant only during the lifetime of the object. Two objects with non-overlapping lifetimes may have the same `id()` value.

Name assignment

Assigning a name to an object provides a way of referencing it, so Python will not delete an object that has a name pointing to it.

In C, Perl, Java and many other languages, we can view variables as names for containers where values are stored. We replace values in the containers with new values via the assignment operator. For example, in Perl:

```
$a = 5;  # the value 5 is saved in the variable $a
$a = 6;  # the value 6 replaces the value 5 in the variable $a
$b = $a; # the contents of the variable $a (6) replaces the value in $b
```

Python is different. Instead of "variables" Python has *names* and names are just references (pointers) to objects. Objects do not persist unless they have a name attached to them. Another way to think about this is that objects that cannot be referenced, have no purpose, and can be deleted. Names are attached to objects via the assignment operator. An assignment statement looks like this:

name = expression

Python evaluates the expression on the right hand side and creates an appropriate object. It then associates the name on the LHS to the object. Note that a name is not part of an object. Instead Python maintains names and their references to objects in tables known as *namespaces*.

To begin appreciating the meaning and implication of these words consider the following:

```
>>> a="foo"
>>> id(a)
4449501016
>>> id(a)
4449501016
```

In the first statement Python creates a string object for "foo" and then attaches the name 'a' to the object. Since the string object that holds "foo" is now associated with the name 'a', it will persist. Accordingly, when we invoke `id()` the name 'a', `id()` now always returns the unique id of the persistent string object.

Up to this point the distinction between names and variables might seem a little pointless, but now consider the following:

```
>>> x = [1,0]
>>> y = x
>>> print y
[1, 0]
>>> x[1] = -1
>>> print x
[1, -1]
>>> print y
[1, -1]
```

If you are still thinking of variables as containers instead of names, then the last line will startle you since it appears that changing one variable changed another. Perhaps the following two lines will shed some light:

```
>>> id(x)
4449382200
>>> id(y)
4449382200
```

These two lines reveal that the two names 'x' and 'y' are both associated with the same object. This was accomplished by the statement `y = x`.

It is now clear what Python is doing. The statement `x[1] = -1` changed the second element of the list whose unique id is 4449382200. We referred to that same list with two names: 'x' *and* 'y'.

If you are a C or perl programmer you will recognize this as the behavior of pointers (in C) or references (in Perl). The object id plays the role of the Perl reference. In perl this would look like this:

```
$x = [1,0];     # create a reference to the list (1,0) and save it in $x
$y = $x;        # copy the reference to $y
print @$y;      # dereference the array reference and print it
$x->[1] = -1;   # save -1 in the second element of the list
print @$y;      # dereference the array reference and print it
```

To summarize: Effectively, names in Python are references (pointers) to objects. If you internalize that one thought, you will be happy as a clam when you write your code and you will avoid much of the confusion that arises when debugging your Python code.

Objects, methods and attributes

All objects have methods and attributes that have assigned names. Let's use the `dir()` function view the names of the methods and attributes in our string objects:

```
>>> dir("foo") #  and with all these names
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

Let's explore how python works by trying out a couple methods. The syntax for invoking a method is as follows: . <method_name>(). Accordingly, let's try the `upper` method:

```
>>> print("foo".upper()) # convert to upper case
FOO
```

In this statement Python creates a string object to hold the string "foo". Then the `upper` method returns a new object

with the "foo" string converted to upper case. Finally, after the statement is executed, Python marks the two string objects (the one that holds "foo" and the one that holds "FOO") for destruction. The objects are not actually destroyed until Python needs the memory. Now let's try a method that takes an argument:

```
>>> print("foo".__add__("bar")) # __add__ concatenates two str objects
foobar
>>> id( "foo".__add__("bar") ) # the __add__ method returns another object
4449552640
```

In the first statement Python creates two string objects for the "foo" and "bar" strings. Then the `__add__` method of the "foo" object is invoked with the "bar" object as its argument. The method returns a new object with the concatenated string which is then passed to the `print` function. All the same things happen in the third statement except that the returned object is passed to the id() function. All three objects are marked for destruction after each statement.