In this post, I'll help you understand what's happening behind the scenes when you do common things like creating a variable or calling a function. As a result, you'll write cleaner, more comprehensible code. You'll also become a better (and faster) code reader. All that's necessary is to forget everything you know about programming...

"Everything is an object?"

When most people first hear that in Python, "everything is an object", it triggers flashbacks to languages like Java where everything the *user* writes is encapsulated in an object. Others assume this means that in the implementation of the Python interpreter, everything is implemented as objects. The first interpretation is wrong; the second is true but not particularly interesting (for our purposes). What the phrase actually refers to is the fact that all "things", be they values, classes, functions, object instances (obviously), and almost every other language construct is conceptually an object.

What does it mean for everything to be an object? It means all of the "things" mentioned above have all the properties we usually associate with objects (in the object oriented sense); types have member functions, functions have attributes, modules can be passed as arguments, etc. And it has important implications with regards to how assignment in Python works.

A feature of the Python interpreter that often confuses beginners is what happens when print() is called on a "variable" assigned to a user-defined object (I'll explain the quotes in a second). With built-in types, a proper value is usually printed, like when calling print() on strings and ints. For simple, user-defined classes, though, the interpreter spits out some odd looking string like:

>>> **class** **Foo**(): **pass**

>>> foo = Foo()

>>> **print**(foo)

<\_\_main\_\_.Foo object at 0xd3adb33f]]]]>

print() is supposed to print the value of a "variable", right? So why is it printing that garbage?

To answer that, we need to understand what foo actually represents in Python. Most other languages would call it a variable. Indeed, many Python articles would refer to foo as a variable, but really only as a shorthand notation.

In languages like C, foo represents storage for "stuff". If we wrote

int foo = 42;

it would be correct to say that the integer variable foo contained the value 42. That is, *variables are a sort of container for values*.

And now for something completely different...

In Python, this isn't the case. When we say:

>>> foo = Foo()

it would be wrong to say that foo "contained" a Foo object. Rather, foo *is a name with a binding to the objectcreated by* Foo(). The portion of the right hand side of the equals sign creates an object. Assigning foo to that object merely says "I want to be able to refer to this object as foo." **Instead of variables (in the classic sense), Python has names and bindings**.

So when we printed foo earlier, what the interpreter was showing us was the address in memory where the object that foo is bound to is stored. This isn't as useless as it sounds. If you're in the interpreter and want to see if two names are bound to the same object, you can do a quick-and-dirty check by printing them and comparing the addresses. If they match, they're bound to the same object; if not, their bound to different objects. Of course, the idiomatic way to check if two names are bound to the same object is to use is

If we continued our example and wrote

>>> baz = foo

we should read this as "Bind the name baz to the same object foo is bound to (whatever that may be)." It should be clear, then why the following happens

>>> baz.some\_attribute

Traceback (most recent call last):

File "<stdin>", line 1, **in** <module]]]]>

**AttributeError**: 'Foo' object has no attribute 'some\_attribute'

>>> foo.some\_attribute = 'set from foo'

>>> baz.some\_attribute

'set from foo'

Changing the object in some way using foo will also be reflected in baz: they are both bound to the same underlying object.

What's in a name...

names in Python are not unlike names in the real world. If my wife calls me "Jeff", my dad calls me "Jeffrey", and my boss calls me "Idiot", it doesn't fundamentally change *me*. If my boss decides to call me "Captain Programming," great, but it still hasn't changed anything about me. It does mean, however, that if my wife kills "Jeff" (and who could blame her), "Captain Programming" is also dead. Likewise, in Python binding a name to an object doesn't change it. Changing some property of the object, however, will be reflected in all other names bound to that object.

Everything really *is* an object. I swear.

Here, a questions arises: How do we know that the thing on the right hand side of the equals sign will always be an object we can bind a name to? What about

>>> foo = 10

or

>>> foo = "Hello World!"

Now is when "everything is an object" pays off. Anything you can (legally) place on the right hand side of the equals sign is (or creates) an object in Python. Both 10 and Hello World are objects. Don't believe me? Check for yourself

>>> foo = 10

>>> **print**(foo.\_\_add\_\_)

<method-wrapper '\_\_add\_\_' of int object at 0x8502c0]]]]>

If 10 was actually just the number '10', it probably wouldn't have an \_\_add\_\_ attribute (or any attributes at all).

In fact, we can see all the attributes 10 has using the dir() function:

>>> dir(10)

['\_\_abs\_\_', '\_\_add\_\_', '\_\_and\_\_', '\_\_class\_\_', '\_\_cmp\_\_', '\_\_coerce\_\_', '\_\_delattr\_\_',

'\_\_div\_\_', '\_\_divmod\_\_', '\_\_doc\_\_', '\_\_float\_\_', '\_\_floordiv\_\_', '\_\_format\_\_',

'\_\_getattribute\_\_', '\_\_getnewargs\_\_', '\_\_hash\_\_', '\_\_hex\_\_', '\_\_index\_\_',

'\_\_init\_\_', '\_\_int\_\_', '\_\_invert\_\_', '\_\_long\_\_', '\_\_lshift\_\_', '\_\_mod\_\_',

'\_\_mul\_\_', '\_\_neg\_\_', '\_\_new\_\_', '\_\_nonzero\_\_', '\_\_oct\_\_', '\_\_or\_\_',

'\_\_pos\_\_', '\_\_pow\_\_', '\_\_radd\_\_', '\_\_rand\_\_', '\_\_rdiv\_\_', '\_\_rdivmod\_\_',

'\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rfloordiv\_\_', '\_\_rlshift\_\_',

'\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_ror\_\_', '\_\_rpow\_\_', '\_\_rrshift\_\_', '\_\_rshift\_\_',

'\_\_rsub\_\_', '\_\_rtruediv\_\_', '\_\_rxor\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_',

'\_\_sub\_\_', '\_\_subclasshook\_\_', '\_\_truediv\_\_', '\_\_trunc\_\_', '\_\_xor\_\_',

'bit\_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']

With all those attributes and member functions, I think it's safe to say 10 is an object.

Since everything in Python is essentially names bound to objects, we can do silly (but interesting) stuff like this:

>>> **import** **datetime**

>>> **import** **imp**

>>> datetime.datetime.now()

datetime.datetime(2013, 02, 14, 02, 53, 59, 608842)

>>> **class** **PartyTime**():

...     **def** \_\_call\_\_(self, \*args):

...         imp.reload(datetime)

...         value = datetime.datetime(\*args)

...         datetime.datetime = self

...         **return** value

...

...     **def** \_\_getattr\_\_(self, value):

...         **if** value == 'now':

...             **return** **lambda**: **print**('Party Time!')

...         **else**:

...             imp.reload(datetime)

...             value = getattr(datetime.datetime, value)

...             datetime.datetime = self

...             **return** value

>>> datetime.datetime = PartyTime()

>>> datetime.datetime.now()

Party Time!

>>> today = datetime.datetime(2013, 2, 14)

>>> **print**(today)

2013-02-14 00:00:00

>>> **print**(today.timestamp())

1360818000.0

datetime.datetime is just a name (that happens to be bound to an object representing the datetime class). We can rebind it to whatever we please. In the example above, we bind the datetime attribute of the datetime module to our new class, PartyTime. Any call to the datetime.datetime constructor returns a valid datetime object. In fact, the class is indistinguishable from the real datetime.datetime class. Except, that is, for the fact that if you calldatetime.datetime.now() it always prints out 'Party Time!'.

Obviously this is a silly example, but hopefully it gives you some insight into what is possible when you fully understand and make use of Python's execution model. At this point, though, we've only changed bindings associated with a name. What about changing the object itself?

Two types of objects

It turns out Python has two flavors of objects: mutable and immutable. The value of mutable objects can be changed after they are created. The value of immutable objects cannot be. A list is a mutable object. You can create a list, append some values, and the list is updated in place. A string is immutable. Once you create a string, you can't change its value.

I know what you're thinking: "Of course you can change the value of a string, I do it all the time in my code!" When you "change" a string, you're actually rebinding it to a newly created string object. The original object remains unchanged, even though its possible that nothing refers to it anymore.

See for yourself:

>>> a = 'foo'

>>> a

'foo'

>>> b = a

>>> a += 'bar'

>>> a

'foobar'

>>> b

'foo'

Even though we're using += and it *seems* that we're modifying the string, we really just get a new one containing the result of the change. This is why you may hear people say, "string concatenation is slow.". It's because concatenating strings must allocate memory for a new string and copy the contents, while appending to a list (in most cases) requires no allocation. Immutable objects are fundamentally expensive to "change", because doing so involves creating a copy. Changing mutable objects is cheap.

Immutable object weirdness

When I said the value of immutable objects can't change after they're created, it wasn't the whole truth. A number of containers in Python, such as tuple, are immutable. The value of a tuple can't be changed after it is created. But the "value" of a tuple is conceptually just a sequence of names with unchangeable bindings to objects. The key thing to note is that the *bindings* are unchangeable, not the objects they are bound to.

This means the following is perfectly legal:

>>> **class** **Foo**():

...     **def** \_\_init\_\_(self):

...             self.value = 0

...     **def** \_\_str\_\_(self):

...             **return** str(self.value)

...     **def** \_\_repr\_\_(self):

...             **return** str(self.value)

...

>>> f = Foo()

>>> **print**(f)

0

>>> foo\_tuple = (f, f)

>>> **print**(foo\_tuple)

(0, 0)

>>> foo\_tuple[0] = 100

Traceback (most recent call last):

File "<stdin>", line 1, **in** <module]]]]>

**TypeError**: 'tuple' object does **not** support item assignment

>>> f.value = 999

>>> **print**(f)

999

>>> **print**(foo\_tuple)

(999, 999)

When we try to change an element of the tuple directly, we get a TypeError telling us that (once created), tuplescan't be assigned to. But changing the underlying object has the effect of "changing" the value of the tuple. This is a subtle point, but nonetheless important: the "value" of an immutable object *can't* change, but it's constituent objects*can*.

Function calls

If variables are just names bound to objects, what happens when we pass them as arguments to a function? The truth is, we aren't really passing all that much. Take a look at this code:

**def** add\_to\_tree(root, value\_string):

*"""Given a string of characters `value\_string`, create or update a*

*series of dictionaries where the value at each level is a dictionary of*

*the characters that have been seen following the current character.*

*Example:*

*>>> my\_string = 'abc'*

*>>> tree = {}*

*>>> add\_to\_tree(tree, my\_string)*

*>>> print(tree['a']['b'])*

*{'c': {}}*

*>>> add\_to\_tree(tree, 'abd')*

*>>> print(tree['a']['b'])*

*{'c': {}, 'd': {}}*

*>>> print(tree['a']['d'])*

*KeyError 'd'*

*"""*

**for** character **in** value\_string:

        root = root.setdefault(character, {})

We're essentially creating an auto-vivifying dictionary that operates like a trie. Notice that we change the rootparameter in the for loop. And yet after the function call completes, tree is still the same dictionary with some updates. It is *not* the last value of root in the function call. So in one sense tree is being updated; in another sense it's not.

To make sense of this, consider what the root parameter actually is: a *new* binding to the object refereed to by the name passed in as the root parameter. In the case of our example, root is a name initially bound to the same object as tree. It is *not* tree itself, which explains why changing root to a new dictionary in the function leavestree unchanged. As you'll recall, assigning root to root.setdefault(character, {}) merely rebinds root to the object created by the root.setdefault(character, {}) statement.

Here's another, more straightforward, example:

**def** list\_changer(input\_list):

    input\_list[0] = 10

    input\_list = range(1, 10)

**print**(input\_list)

    input\_list[0] = 10

**print**(input\_list)

>>> test\_list = [5, 5, 5]

>>>

[1, 2, 3, 4, 5, 6, 7, 8, 9]

[10, 2, 3, 4, 5, 6, 7, 8, 9]

>>> **print** test\_list

[10, 5, 5]

Our first statement *does* change the value of the underlying list (as we can see in the last line printed). However, once we rebind the name input\_list by saying input\_list = range(1, 10), **we're now referring to a completely different object**. We basically said "bind the name input\_list to this new list." After that line, we have no way of referring to the original input\_list parameter again.

By now, you should have a clear understanding of how binding a name works. There's just one more item to take care of.

Blocks and Scope

The concepts of names, bindings, and objects should be quite familiar at this point. What we haven't covered, though, is how the interpreter "finds" a name. To see what I mean, consider the following:

GLOBAL\_CONSTANT = 42

**def** print\_some\_weird\_calculation(value):

    number\_of\_digits = len(str(value))

**def** print\_formatted\_calculation(result):

**print**('{value} \* {constant} = {result}'.format(value=value,

            constant=GLOBAL\_CONSTANT, result=result))

**print**('{}   {}'.format('^' \* number\_of\_digits, '++'))

**print**('**\n**Key: ^ points to your number, + points to constant')

    print\_formatted\_calculation(value \* GLOBAL\_CONSTANT)

>>> print\_some\_weird\_calculation(123)

123 \* 42 = 5166

^^^   ++

Key: ^ points to your number, + points to constant

This is a contrived example, but a couple of things should jump out at you. First, how does theprint\_formatted\_calculation function have access to value and number\_of\_digits even though they were never passed as arguments? Second, how do both functions seem to have access to GLOBAL\_CONSTANT?

The answer is all about scope. In Python, when a name is bound to an object, that name is only usable within the name's scope. The scope of a name is determined by the block in which it was created. A block is just a "block" of Python code that is executed as a single unit. The three most common types of blocks are modules, class definitions, and the bodies of functions. So the scope of a name is the innermost block in which it's defined.

Let's now return to the original question: how does the interpreter "find" what a name is bound to (or if it's even a valid name at all)? It begins by checking the scope of the innermost block. Then it checks the scope that contained the innermost block, then the scope that contained that, and so on.

In the print\_formatted\_calculation function, we reference value. This is resolved by first checking the scopeof the innermost block, which in this case is the body of the function itself. When it doesn't find value defined there, it checks the scope that print\_formatted\_calculation was defined in. In our case, that's the body of theprint\_some\_weird\_calculation function. Here it does find the name value, and so it uses that binding and stops looking. The same is true for GLOBAL\_CONSTANT, it just needs to look an extra level higher: the module (or script) level. Anything defined at this level is considered a global name. These are accessible from anywhere.

A few quick things to note. A name's scope extends to any blocks contained in the block where the name was defined, *unless the name is rebound in one of those blocks*. If print\_formatted\_calculation had the line value = 3, then the scope of the name value in print\_some\_weird\_calculation would only be the body of that function. It's scope would not include print\_formatted\_calculation, since that block rebound the name.

Use this power wisely...

There are two keywords that can be used to tell the interpreter to **reuse a preexisting binding**. Every other time we bind a name, it binds that name to a new object, *but only in the current scope*. In the example above, if we reboundvalue in print\_formatted\_calculation, it would have no affect on the value inprint\_some\_weird\_calculation, which is print\_formatted\_calculation's enclosing scope. With the following two keywords, we can actually affect the bindings outside our local scope.

global my\_variable tells the interpreter to use the binding of the name my\_variable in the top-most (or "global" scope). Putting global my\_variable in a code block is a way of saying, "copy the binding of this global variable, or if you don't find it, create the name my\_variable in the global scope." Similarly, the nonlocal my\_variablestatement instructs the interpreter to use the binding of the name my\_variable defined in the nearest *enclosing*scope. This is a way to rebind a name not defined in either the local or global scope. Without nonlocal, we would only be able to alter bindings in the local scope or the global scope. Unlike global my\_variable however, if we usenonlocal my\_variable then my\_variable must already exist; it won't be created if it's not found.

To see this in action, let's write a quick example:

GLOBAL\_CONSTANT = 42

**print**(GLOBAL\_CONSTANT)

**def** outer\_scope\_function():

    some\_value = hex(0x0)

**print**(some\_value)

**def** inner\_scope\_function():

        nonlocal some\_value

        some\_value = hex(0xDEADBEEF)

    inner\_scope\_function()

**print**(some\_value)

**global** GLOBAL\_CONSTANT

    GLOBAL\_CONSTANT = 31337

outer\_scope\_function()

**print**(GLOBAL\_CONSTANT)

*# Output:*

*# 42*

*# 0x0*

*# 0xdeadbeef*

*# 31337*

By making use of global and nonlocal, we're able to use and change the existing binding of a name rather than merely assigning the name a new binding and losing the old one.

———————— X ———————— X ————————