Facts and myths about Python names and values

Created 6 July 2013, last updated 26 January 2014

The behavior of names and values in Python can be confusing. Like many parts of Python, it has an underlying simplicity that can be hard to discern, especially if you are used to other programming languages. Here I'll explain how it all works, and present some facts and myths along the way.

BTW: I worked this up into a presentation for PyCon 2015: Python Names and Values.

Names and values

Let's start simple:

> Fact: Names refer to values.

As in many programming languages, a Python assignment statement associates a symbolic name on the left-hand side with a value on the right-hand side. In Python, we say that names refer to values, or a name is a reference to a value:

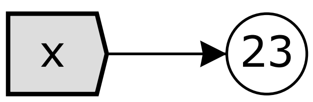
```
x = 23
```

Now the name "x" refers to the value 23. The next time we use the name x, we'll get the value 23:

```
print(x+2)       # prints 25
```

Exactly how the name refers to the value isn't really important. If you're experienced with the C language, you might like to think of it as a pointer, but if that means nothing to you then don't worry about it.

To help explain what's going on, I'll use diagrams. A gray rectangular tag-like shape is a name, with an arrow pointing to its value. Here's the name x referring to an integer 23:



I'll be using these diagrams to show how Python statements affect the names and values involved. (The diagrams are SVG, if they don't render, let me know.)

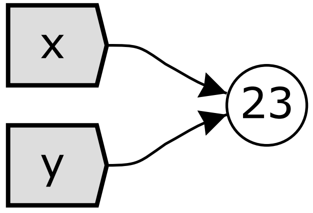
Another way to explore what's going on with these code snippets is to try them on pythontutor.com, which cleverly diagrams your code as it runs. I've included links there with some of the examples.

> Fact: Many names can refer to one value.

There's no rule that says a value can only have one name. An assignment statement can make a second (or third, ...) name refer to the same value.

```
x = 23
y = x
```

Now x and y both refer to the same value:

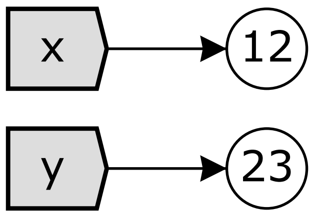


Neither x or y is the "real" name. They have equal status: each refers to the value in exactly the same way.

> Fact: Names are reassigned independently of other names.

If two names refer to the same value, this doesn't magically link the two names. Reassigning one of them won't reassign the other also:

```
x = 23
y = x
x = 12
```



When we said "y = x", that doesn't mean that they will always be the same forever. Reassigning x leaves y alone. Imagine the chaos if it didn't!

> Fact: Values live until nothing references them.

Python keeps track of how many references each value has, and automatically cleans up values that have none. This is called "garbage collection," and means that you don't have to get rid of values, they go away by themselves when they are no longer needed.

Exactly how Python keeps track is an implementation detail, but if you hear the term "reference counting," that's an important part of it. Sometimes cleaning up a value is called reclaiming it.

Assignment

An important fact about assignment:

> Fact: Assignment never copies data.

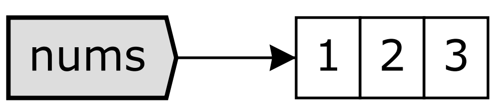
When values have more than one name, it's easy to get confused and think of it as two names and two values:

```
x = 23
y = x
# "Now I have two values: x and y!"
# NO: you have two names, but only one value.
```

Assigning a value to a name never copies the data, it never makes a new value. Assignment just makes the name on the left refer to the value on the right. In this case, we have only one 23, and x and y both refer to it, just as we saw in the last diagrams.

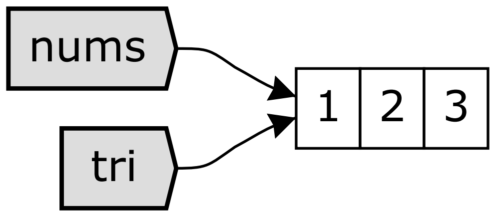
Things get more interesting when we have more complicated values, like a list:

```
nums = [1, 2, 3]
```



Now if we assign nums to another name, we'll have two names referring to the same list:

```
nums = [1, 2, 3]
tri = nums
```



Remember: assignment never makes new values, and it never copies data. This assignment statement doesn't magically turn my list into two lists.

At this point, we have one list, referred to by two names, which can lead to a big surprise which is common enough I'm going to give it a catchy name: the Mutable Presto-Chango.

> Fact: Changes in a value are visible through all of its names. (Mutable Presto-Chango)

Values fall into two categories based on their type: mutable or immutable. Immutable values include numbers, strings, and tuples. Almost everything else is mutable, including lists, dicts, and user-defined objects. Mutable means that the value has methods that can change the value in-place. Immutable means that the value can never change, instead when you think you are changing the value, you are really making new values from old ones.

Since numbers are immutable, you can't change one in-place, you can only make a new value and assign it to the same name:

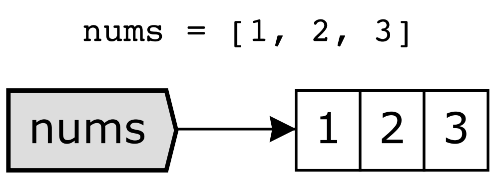
```
x = 1
x = x + 1
```

Here, x+1 computes an entirely new value, which is then assigned to x.

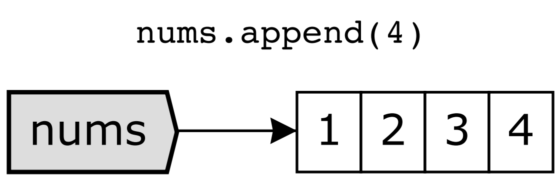
With a mutable value, you can change the value directly, usually with a method on the value:

```
nums = [1, 2, 3]
nums.append(4)
```

First we assign a list to a name:



Then we append another value onto the list:



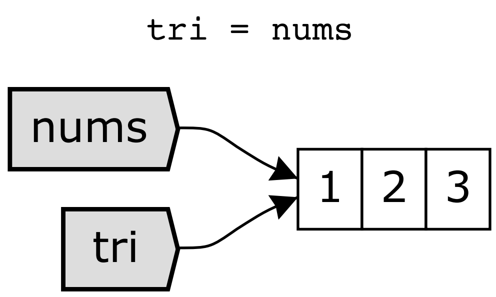
Here we haven't changed which value nums refers to. At first, the name nums refers to a three-element list. Then we use the name nums to access the list, but we don't assign to nums, so the name continues to refer to the same list. The append method modifies that list by appending 4 to it, but it's the same list, and nums still refers to it. This distinction between assigning a name and changing a value is sometimes described as "rebinding the name vs. mutating the value."

Notice that informal English descriptions can be ambigious. We might say that "x = x+1" is changing x, and "nums.append(4)" is changing nums, but they are very different kinds of change. The first makes x refer to a new value (rebinding), the second is modifying the value x refers to (mutating).

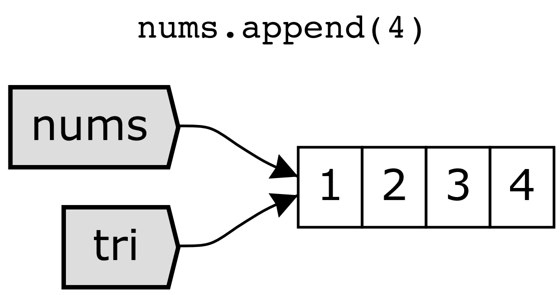
Here's where people get surprised: if two names refer to the same value, and the value is mutated, then both names see the change:

```
nums = [1, 2, 3]
tri = nums
nums.append(4)

print(tri)      # [1, 2, 3, 4]
```

Why did tri change!? The answer follows from what we've learned so far. Assignment never copies values, so after the assignment to tri, we have two names referring to the same list:



Then we mutate the list by calling .append(4), which modifies the list in place. Since tri refers to that list, when we look at tri we see the same list as nums, which has been changed, so tri now shows four numbers also:



This Mutable Presto-Chango is the biggest issue people have with Python's names and values. A value is shared by more than one name, and is modified, and all names see the change. To make the Presto-Chango happen, you need:

- A mutable value, in this case the list,
- More than one name referring to the value,
- Some code changes the value through one of the names, and
- The other names see the change.

Keep in mind, this is not a bug in Python, however much you might wish that it worked differently. Many values have more than one name at certain points in your program, and it's perfectly fine to mutate values and have all the names see the change. The alternative would be for assignment to copy values, and that would make your programs unbearably slow.

> Myth: Python assigns mutable and immutable values differently.

Because the Presto-Chango only happens with mutable values, some people believe that assignment works differently for mutable values than for immutable values. It doesn't.

All assignment works the same: it makes a name refer to a value. But with an immutable value, no matter how many names are referring to the same value, the value can't be changed in-place, so you can never get into a surprising Presto-Chango situation.

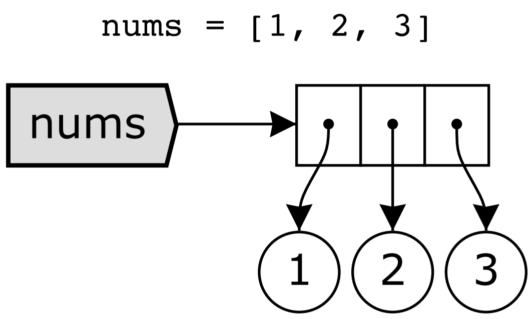
Python's diversity

I said earlier that Python has an underlying simplicity. Its mechanisms are quite simple, but they manifest in a number of ways.

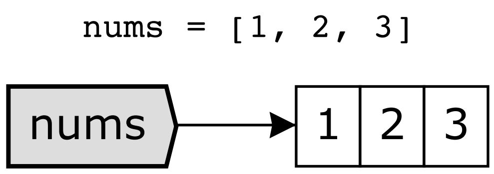
Fact: References can be more than just names.

All of the examples I've been using so far used names as references to values, but other things can be references. Python has a number of compound data structures each of which hold references to values: list elements, dictionary keys and values, object attributes, and so on. Each of those can be used on the left-hand side of an assignment, and all the details I've been talking about apply to them. Anything that can appear on the left-hand side of an assignment statement is a reference, and everywhere I say "name" you can substitute "reference".

In our diagrams of lists, I've shown numbers as the elements, but really, each element is a reference to a number, so it should be drawn like this:



But that gets complicated quickly, so I've used a visual shorthand:



If you have list elements referring to other mutable values, like sub-lists, it's important to remember that the list elements are just references to values.

Here are some other assignments. Each of these left-hand sides is a reference:

```
my_obj.attr = 23
my_dict[key] = 24
my_list[index] = 25
my_obj.attr[key][index].attr = "etc, etc"
```

and so on. Lots of Python data structures hold values, and each of those is a reference. All of the rules here about names apply exactly the same to any of these references. For example, the garbage collector doesn't just count names, it counts any kind of reference to decide when a value can be reclaimed.

Note that "i = x" assigns to the name i, but "i[0] = x" doesn't, it assigns to the first element of i's value. It's important to keep straight what exactly is being assigned to. Just because a name appears somewhere on the left-hand side of the assignment statement doesn't mean the name is being rebound.

Fact: Lots of things are assignment

Just as many things can serve as references, there are many operations in Python that are assignments. Each of these lines is an assignment to the name X:

```
X = ...
for X in ...
[... for X in ...]
(... for X in ...)
{... for X in ...}
class X(...):
```

```
def X(...):
def fn(X): ... ; fn(12)
with ... as X:
except ... as X:
import X
from ... import X
import ... as X
from ... import ... as X
```

I don't mean that these statements act kind of like assignments. I mean that these are assignments: they all make the name X refer to a value, and everything I've been saying about assignments applies to all of them uniformly.

For the most part, these statements define X in the same scope as the statement, but not all of them, especially the comprehensions, and the details differ slightly between Python 2 and Python 3. But they are all real assignments, and every fact about assignment applies to all of them.

> Fact: Python passes function arguments by assigning to them.

Let's examine the most interesting of these alternate assignments: calling a function. When I define a function, I name its parameters:

```
def my_func(x, y):
    return x+y
```

Here x and y are the parameters of the function my_func. When I call my_func, I provide actual values to be used as the arguments of the function. These values are assigned to the parameter names just as if an assignment statement had been used:

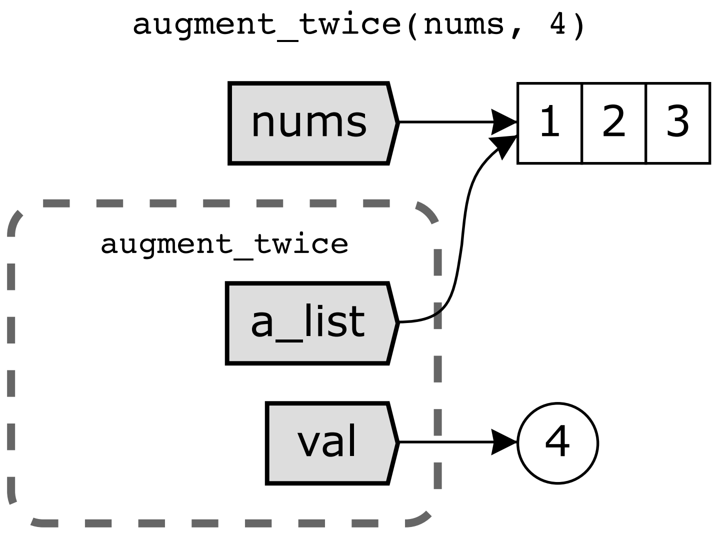
```
def my_func(x, y)
    return x+y

print(my_func(8, 9))
```

When my_func is called, the name x has 8 assigned to it, and the name y has 9 assigned to it. That assignment works exactly the same as the simple assignment statements we've been talking about. The names x and y are local to the function, so when the function returns, those names go away. But if the values they refer to are still referenced by other names, the values live on.

Just like every other assignment, mutable values can be passed into functions, and changes to the value will be visible through all of its names:

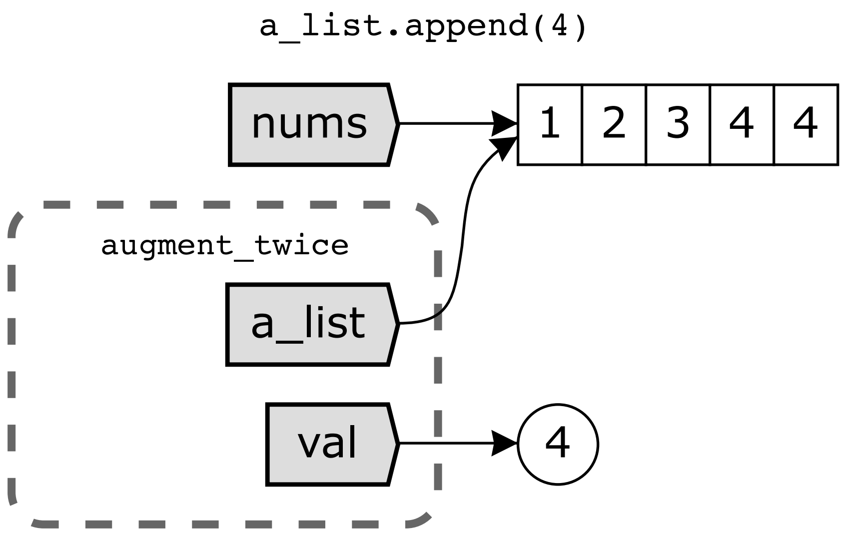
```
def augment_twice(a_list, val):
    """Put `val` on the end of `a_list` twice."""
    a_list.append(val)
    a_list.append(val)

nums = [1, 2, 3]
augment_twice(nums, 4)
print(nums)         # [1, 2, 3, 4, 4]
```

This can produce surprising results, so let's take this step by step. When we call augment_twice, the names and values look like this:

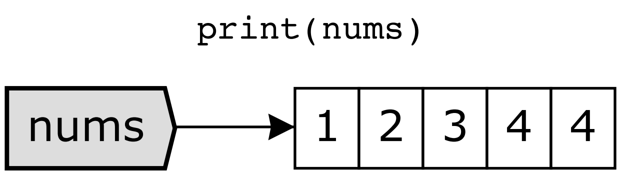


The local names in the function are drawn in a new frame. Calling the function assigned the actual values to the parameter names, just like any other assignment statement. Remember that assignment never makes new values or copies any data, so the here the local name a_list refers to the same value that was passed in, nums.

Then we call a_list.append twice, which mutates the list:



When the function ends, the local names are destroyed. Values that are no longer referenced are reclaimed, but others remain:



You can try this example code yourself [on pythontutor.com]().

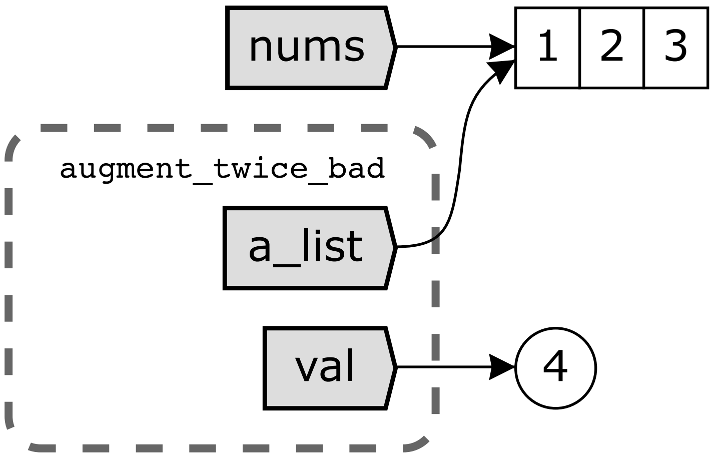
We passed the list into the function, which modified it. No values were copied. Although this behavior might be surprising, it's essential. Without it, we couldn't write methods that modify objects.

Here's another way to write the function, but it doesn't work. Let's see why.

```
def augment_twice_bad(a_list, val):
    """Put `val` on the end of `a_list` twice."""
    a_list = a_list + [val, val]

nums = [1, 2, 3]
augment_twice_bad(nums, 4)
print(nums)         # [1, 2, 3]
```

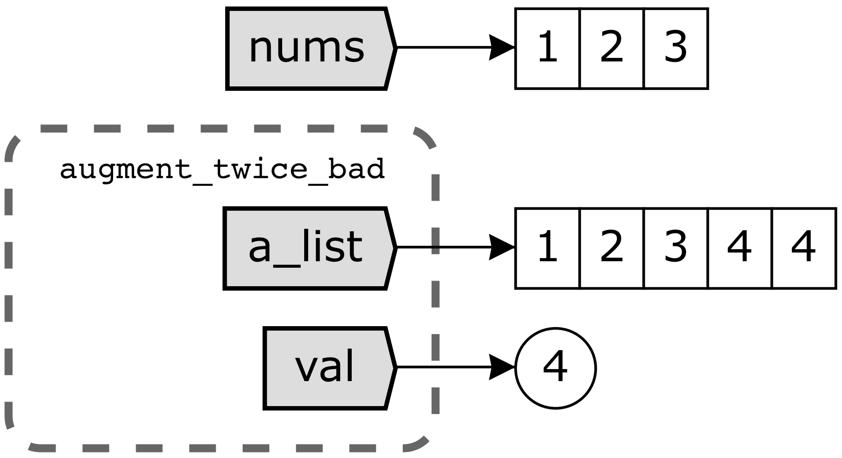
At the moment we call augment_twice_bad, it looks the same as we saw earlier with augment_twice:

`augment_twice_bad(nums, 4)`



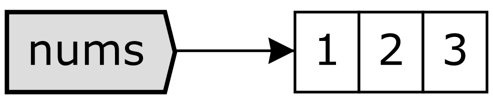
The next statement is an assignment. The expression on the right-hand side makes a new list, which is then assigned to a_list:

`a_list = a_list + [val, val]`



When the function ends, its local names are destroyed, and any values no longer referenced are reclaimed, leaving us just where we started:

`print(nums)`



(Try this code on pythontutor.com.)

It's really important to keep in mind the difference between mutating a value in place, and rebinding a name. augment_twice worked because it mutated the value passed in, so that mutation was available after the function returned. augment_twice_bad used an assignment to rebind a local name, so the changes weren't visible outside the function.

Another option for our function is to make a new value, and return it:

```
def augment_twice_good(a_list, val):
    a_list = a_list + [val, val]
    return a_list

nums = [1, 2, 3]
nums = augment_twice_good(nums, 4)
print(nums)          # [1, 2, 3, 4, 4]
```

Here we make an entirely new value inside augment_twice_good, and return it from the function. The caller uses an assignment to hold onto that value, and we get the effect we want.

This last function is perhaps the best, since it creates the fewest surprises. It avoids the Presto-Chango by not mutating a value in-place, and only creating new values.

There's no right answer to choosing between mutating and rebinding: which you use depends on the effect you need. The important thing is to understand how each behaves, to know what tools you have at your disposal, and then to pick the one that works best for your particular problem.

Dynamic typing

Some details about Python names and values:

> Fact: Any name can refer to any value at any time.

Python is dynamically typed, which means that names have no type. Any name can refer to any value at any time. A name can refer to an integer, and then to a string, and then to a function, and then to a module. Of course, this could be a very confusing program, and you shouldn't do it, but the Python language won't mind.

> Fact: Names have no type, values have no scope.

Just as names have no type, values have no scope. When we say that a function has a local variable, we mean that the name is scoped to the function: you can't use the name outside the function, and when the function returns, the name is destroyed. But as we've seen, if the name's value has other references, it will live on beyond the function call. It is a local name, not a local value.

> Fact: Values can't be deleted, only names can.

Python's memory management is so central to its behavior, not only do you not have to delete values, but there is no way to delete values. You may have seen the del statement:

```
nums = [1, 2, 3]
del nums
```

This does not delete the value nums, it deletes the name nums. The name is removed from its scope, and then the usual reference counting kicks in: if nums' value had only that one reference, then the value will be reclaimed. But if it had other references, then it will not.

> Myth: Python has no variables.

Some people like to say, "Python has no variables, it has names." This slogan is misleading. The truth is that Python has variables, they just work differently than variables in C.

Names are Python's variables: they refer to values, and those values can change (vary) over the course of your program. Just because another language (albeit an important one) behaves differently is no reason to describe Python as not having variables.

Wrapping up

> Myth? Python is confusing.

I hope this has helped clarify how names and values work in Python. It's a very simple mechanism that can be surprising, but is very powerful. Especially if you are used to languages like C, you'll have to think about your values differently.

There are lots of side trips that I skipped here:

- Is Python call-by-value or not?
- Why do beginners find it hard to make a tic-tac-toe board in Python? Answered in [Names and values: making a game board](#).
- Why is "list += seq" not the same as "list = list + seq"?
- Why is "is" different than "==" and how come "2 + 2 is 4", but "1000 + 1 is not 1001"?

- What's the deal with mutable default arguments to functions?
- Why is it easy to make a list class attribute, but hard to make an int class attribute?

See also

If you are looking for more information about these topics, try:

- Python Tutor, which visualizes program execution, including bindings of names to values.
- How to Think Like a Pythonista, which explains all this, with ASCII art!
- My blog, where posts occasionally appear about these sorts of things.

Comments

Daniel Biro 2:46 PM on 8 Jul 2013

Hello!

These are good explanations. Make complicated things simple by using proper and clear abstractions. You are a good writer.

I would like to see an explanation of the execution model on the skip list.
The documentation is hard to understand and full of loose ends. I am not alone with this.

http://docs.python.org/2/reference/executionmodel.html
http://bugs.python.org/issue12374

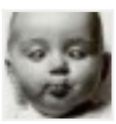
This is an attempt by me to give an explanation for it, on apropos of the default mutable value:
http://www.daniweb.com/software-development/python/threads/457567/somebody-please-explain-why-this-is-happening#post1990653

Bye
Daniel

Ned Batchelder 3:06 PM on 8 Jul 2013

@Daniel, thanks. Your issue is already on the skip list: "What's the deal with mutable default arguments to functions?"

In a nutshell: default values for arguments are evaluated just once, when the function is defined. When the function is called, if a value isn't supplied for that argument, then the default value is assigned to the local name, and execution proceeds. If you mutate that value, then you have a Presto-Chango, where the two references are: 1) the default for the function argument, and 2) the local parameter name. Since values live until there are no more references, and the function object continues to refer to the default value, that mutable default value will live a very long time, and will collect mutations as it goes.



Robert 3:11 AM on 9 Jul 2013

Ned
Can you explain if n and n2 refer to same list ?
If Y: why no mutable-presto-changeo
If N: why so ? why python creates 2 lists here ?

```
>>> n=[1,2,3]
>>> n2=[1,2,3]
>>> n[0]='x'
>>> n,n2
(['x', 2, 3], [1, 2, 3])
```

Matt 5:22 AM on 9 Jul 2013

Robert:

Take Ned's advce and try this on pythontutor.com.

You'll see that the creation of both 'n' and 'n2' create separate lists with the same value. Why? Python could get impossibly slow if it had to seach *every single value* for equality every time a name was assigned a new value. Imagine you have a program that holds 100,000 dictionaries each with 50,000 keys and you start to see the point.

If your code had said 'n2=n', then Python says, "Oh, I already know that value; I'll just create a new name to point to it."

More to the point, Python does this when *you* assign a value to a name via another name. It would hardly be sporting if you had a list of value [1,2,3], and so did some code in a third-party module, and it changed the list for its own purpose. If Python checked all values before assigning names to avoid duplicates, then anyone changing that list would change your list! Insanity would rule. Black is down, up is north, and the sun would set in liters.

You can wind up assigning many names to one value, but every time you assign by value (and not by name), you by necessity create a new instance of the value.

Or so I believe.

Ned Batchelder 11:22 AM on 9 Jul 2013

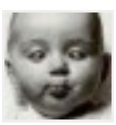
@Robert: Matt has provided a good explanation, I'll add just a bit more. The semantics of Python are that [1,2,3] creates a list with three elements. It always does, no matter how many lists have already been created. Once you've executed

```
n = [1,2,3]
```

these two statements have very different effects:

```
n2 = [1,2,3]    # make a list and assign it to n2
n3 = n          # assign n's value to n3
```

All three names refer to a three-element list, but n and n3 refer to the same list, n2 refers to a different list that happens to be equal.



Robert 2:21 PM on 9 Jul 2013

Thanks, Matt and Ned

So at the "root" - this operation:

aName = aValue

it goes back to whether aValue is mutable or not, right ?
If non-mutable: Same value can be pointed to by multiple names

if mutable: Python just creates aValue right there.

Is this right ?

Ned Batchelder 3:49 PM on 9 Jul 2013

@Robert. Mutable and immutable values are treated exactly the same, refer to the Myth about that very topic above. The expression on the right-hand side of the assignment statement produces a value. How it does that is up to it. A list literal *always* makes a new list. An integer literal might make a new int, or it might not. Python can share integer objects because they are immutable, so there's no way to change their values, so there's no harm in sharing them widely.

Myth: Python assigns mutable and immutable values differently.

Norman 5:46 PM on 11 Jul 2013

> how come "2 + 2 is 4", but "1000 + 1 is not 1001"

I had to try in shell before I believed. Are small ints (not sure of correct terminology) pre-allocated, interned?

Ned Batchelder 5:55 PM on 11 Jul 2013

@Norman: indeed, CPython interns small ints (-5 to 256), and since they are immutable, it doesn't affect the Python semantics.

atul 4:58 AM on 13 Jul 2013

Hi Ned,
Thanks for the article... It made very good reading. Initially I got confused with "2 + 2 is 4", but "1000 + 1 is not 1001" - then I realized the "is" is like reference comparison - and == is value comparison
- similar to java's == versus equals(...)
I think that is what you meant by the interns above...

Thanks - and it was enlightening - learnt something new...

vasu 4:23 PM on 14 Jul 2013

I may be wrong but I think there's a mistake

The first example where
x=23
y=x

The value of 23 is COPIED and assigned to 'y'. So actually you have 2 different values, because when you change the value of y, the value of x remains the same

y = y+2
x # 23

Ned Batchelder 5:25 PM on 14 Jul 2013

@vasu: You are mistaken. The diagrams are correct. Initially x and y refer to the same value. Then the statement "y = y+2" makes y refer to an entirely new value, 25. x is left referring to the old value, 23.

Alan Franzoni 9:42 PM on 14 Jul 2013

Nice article! Good and thorough explanation of the way Python handles names and values.

One thing I don't agree with: I really think that the "Python has no variables" makes things far easier for the novice. Saying that variables exist but have a different behaviour makes things much more complicate. "variable" contains a very precise built-in idea of "variable within a certain scope" - normally a specific type - which is got no match in Python. We shouldn't have variable-envy, let's name a different concept in a different way.

Ned Batchelder 10:53 PM on 14 Jul 2013

@Alan: I guess we'll just have to disagree. This is my definition of a variable (taken from Wikipedia): "a symbolic name associated with a value and whose associated value may be changed." That definition applies equally well to C and to Python, without getting into language-specific details.

According to your logic, Javascript has no variables either, which makes explaining the "var" keyword a bit tricky.

Don't let C-myopia make perfectly good terms off-limits to other languages.

Jamie Alexandre 5:52 AM on 15 Jul 2013

Thanks for the great explanations, Ned! I'll definitely be directing students here when they ask me these questions.

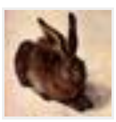
At the risk of seeming pedantic: as I was reading through this, a counter-example came to mind that I thought might be an interesting footnote:
https://gist.github.com/jamalex/5997735

In short: sometimes assignment **does** make new values, if it's been specifically set up to do so. Yay, magic methods!

Ned Batchelder 11:39 AM on 15 Jul 2013

@Jamie, thanks, that's a good point. The complex assignments (attribute, element, item, augmented) can all be overridden with special methods, and therefore can be arbitrarily different from the built-in behavior. There isn't even a guarantee that they will assign anything!

On the other hand, simple name assignment cannot be overridden.



Ankit 8:31 PM on 16 Jul 2013

Hi Ned. I've recently decided that I want to learn Python in-depth. I've written a fair number of scripts with python, but haven't learnt it deeply - e.g I know what generators and decorators are, but never use them, or the fact that lists are mutable types and hence unhashable constantly trips me up (why not have a frozenlist like a frozenset?).

Could you recommend good resources to up my Python-fu? Ideally I'm looking for something comprehensive, like a book/course/workshop/series that I can go through.

Thanks, and nice article. Names and values get tricky because they seem to behave like C most of the time, but they are really quite different.

Aaron Meurer 2:39 AM on 18 Jul 2013

Reading through the facts and myths, and looking at the side trips a the end, I felt like these are all things that I pretty much understood, until I got to the very last one, "Why is it easy to make a list class attribute, but hard to make an int class attribute?" Can you explain what you are referring to here? I've never come across this issue. Do you mean that it is hard to do

class Stuff(object):
a = 1

or

class Stuff(int):
a = 'myattr'

or what? Are you referring to a use-case in the former where instances mutate the class attribute?

Aaron Meurer 2:43 AM on 18 Jul 2013

I find it interesting that you never mention the locals() dictionary, or namespaces in general.

Steve 6:47 PM on 19 Jul 2013

I really like the diagrams in your post. What software did you use to create them?
Thanks

Ned Batchelder 8:33 PM on 19 Jul 2013

@Steve, thanks. I think I had mentioned this in an earlier draft, but it's not in the text here: I used graphviz with some Python code to help make them all uniform.

Ned Batchelder 8:35 PM on 19 Jul 2013

@aaron: the difficulty with class attributes: self.my_int = 2 vs. self.my_list.append("thing")

I didn't mention namespaces because this mostly was not about scoping, and I didn't mention locals() because it's an esoteric attractive nuisance. I left out lots of stuff that's more important than locals()!

Aaron Meurer 9:38 PM on 19 Jul 2013

I see. That's more of an issue with the way Python makes class variables accessible from instances. I think if you do self.__class__.my_int = 2 you should be fine.

Ben 7:03 AM on 26 Jul 2013

Great article, I didn't know that about small ints and the is keyword!

But now you've got me wanting more... Why **do** beginners find it hard to make a tic-tac-toe board in Python??

Aaron Meurer 1:46 PM on 26 Jul 2013

Maybe because they try [[], [], []]*3.

Chris Lee 8:05 PM on 24 Aug 2013

Hi Ned,
one thing that helped me understand these aspects of Python when first learning it was to think "all these things are dictionaries", e.g.

* "names" are a dictionary (ie. NameError is just KeyError from locals() or globals()). The facts you state about assignment etc. follow logically from that.

* object attributes are a dictionary

* function calls are a dictionary (hence args act like assignment... The one subtle exception to this seems to be **kwargs itself, which is always copied rather than assigned in cpython)

So if you understand how Python dictionaries behave, you can understand how all these things behave. Does this seem like a useful viewpoint to you?

Aaron Meurer 9:19 PM on 24 Aug 2013

Be careful. Object attributes are actually much more complicated than a dictionary lookup in the general case.

Ned Batchelder 10:47 PM on 24 Aug 2013

@Chris Lee: I like the analogy of "all things are dictionaries" for understanding object attributes, including modules and classes. I don't see how function calls are dictionaries though, so I wouldn't go as far as you do.

But also, that analogy is a orthongonal to the topics this piece addresses. Once you accept that everything is a dictionary, you still have to understand what d[k]=v does, and that brings you right back to the issues discussed here.

Alok Joshi 4:25 PM on 28 Aug 2013

Super nice

Gordon 7:50 PM on 13 Sep 2013

The diagram after this code block:

```
x = 23
y = x
x = 12
```

seems to be wrong. x should be 12 and y should be 23, but the diagram has them reversed.

Ned Batchelder 9:25 PM on 13 Sep 2013

@Gordon, thanks, I've fixed it.

Rudi Farkas 6:16 PM on 27 Aug 2014

Nice tutorial!
Just for fun, you could add this example:

```
def augment_twice_not_so_bad(a_list, val):
    """Put `val` on the end of `a_list` twice."""
    a_list +=  [val, val]

nums = [1, 2, 3]
augment_twice_not_so_bad(nums, 4)
print(nums)         # [1, 2, 3, 4, 4]
```

sierre 2:20 PM on 13 Jul 2015

Good day Sir, I just wanted to ask if elliptical references are allowed for record types in python such as in namedtuple,dictionaries..etc..?? thanks.