Simulating Pointers in Python

I'm trying to cross compile an in house language(ihl) to Python.

One of the ihl features is pointers and references that behave like you would expect from C or C++.

For instance you can do this:

```
a = [1,2];  // a has an array
b = &a;     // b points to a
*b = 2;     // derefernce b to store 2 in a
print(a);   // outputs 2
print(*b);   // outputs 2
```

Is there a way to duplicate this functionality in Python.

I should point out that I think I've confused a few people. I don't want pointers in Python. I just wanted to get a sense from the Python experts out there, what Python I should generate to simulate the case I've shown above

My Python isn't the greatest but so far my exploration hasn't yielded anything promising:(

I should point out that we are looking to move from our ihl to a more common language so we aren't really tied to Python if someone can suggest another language that may be more suitable.

python pointers

edited Aug 26 '09 at 21:17

asked Jul 17 '09 at 21:13

chollida
5,436 6 44 80

So you're trying to compile a fairly low-level language to a fairly high-level one? Have you considered other platforms for your VM instead? – Nikhil Chelliah Jul 17 '09 at 21:32

3 cluecc.sourceforge.net compiles C to various high level languages, but doesn't have a Python backend yet. It would be interesting to see how it performs, though :) – ephemient Jul 17 '09 at 22:07

One problem with your example is that you are using integers, which in Python is immutable. I.e, you can't change them. This combined with the fact that you have an array variable, which you the overwrite with an integer in your C code (which is horrid from a C perspective) means that you are asking for a way to misuse Python in a way similar to how you misuse C. That just doesn't make any sense. – Lennart Regebro Jul 18 '09 at 7:13

1 @Lennart Regbro. The language I'm coming from isn't C. I thought I'd made that clear to you by now:) – chollida Jul 18 '09 at 17:35

Use size-1 lists for mutable references and non-const pointers: `a=[smth]` `b=[a]` `b[0][0] = 2` `print a[0]` `print b[0][0]` – Dima Tisnek Jan 15 '13 at 20:39

10 Answers

This can be done explicitly.

```
class ref:
    def __init__(self, obj): self.obj = obj
    def get(self):    return self.obj
    def set(self, obj):      self.obj = obj

a = ref([1, 2])
b = a
print a.get()  # => [1, 2]
print b.get()  # => [1, 2]
```

```
b.set(2)
print a.get()  # => 2
print b.get()  # => 2
```

answered Jul 17 '09 at 21:49

ephemient
135k 30 203 334

5 You might not want to use the name "ref", since that's the same name as the weakref reference. Perhaps "ptr" or something. A sensible implementation, though. – Paul Fisher Jul 17 '09 at 21:59

I was thinking of SML, where this is called `'a ref`, but yeah, it would be better to choose a more unique name. Not sure that `ptr` makes all that much sense, though; it's not actually a pointer, it's more like a single container... – ephemient Jul 17 '09 at 22:01

2 Note that this is only even meaningful when using immutable objects like ints or strings. For mutable objects a=Something(); b=a; is perfectly enough. And even with immutable objects it's pretty pointless... – Lennart Regebro Jul 17 '09 at 22:48

3 You can alternately override `__call__` to implement getting and setting, which is more similar to how a weakref behaves. – Miles Jul 18 '09 at 3:08

I really like this, and it is really similar to Python's builtin property function. You could also use ctypes, see my answer. – Mark Mikofski Dec 22 '12 at 7:35

You may want to read *Semantics of Python variable names from a C++ perspective*. The bottom line: **All variables are references**.

More to the point, don't think in terms of variables, but in terms of objects which can be *named*.

edited Jul 17 '09 at 23:43

answered Jul 17 '09 at 21:20

Stephan202
41.6k 6 101 124

3 Of course it has variables; a variable in Python is a named object. Saying "Python doesn't have variables" is just confusing things unnecessarily. – Glenn Maynard Jul 17 '09 at 21:41

@Glenn: I take variable to mean a 'named memory location'. Admittedly, that may not be the correct definition. Though this sentence at Wikipedia, if I interpret it correctly, appears to agree with me: en.wikipedia.org/wiki/Variable_%28programming%29#In_source_code – Stephan202 Jul 17 '09 at 21:49

Not if you are thinking of variables as a fixed in memory space, which is what it is in C. – Lennart Regebro Jul 17 '09 at 21:52

WP says "an identifier that is linked to a value", which I think describes Python just fine. (The specific sentence you linked to doesn't contradict this: it says a variable name is one way to get to a memory location, not that the memory location is the variable.) If you think Python "doesn't have variables", then most other programming languages I've ever used don't, either! :-) – Ken Jul 17 '09 at 23:35

Alright, I changed the wording. Hopefully it's now less controversial :) – Stephan202 Jul 17 '09 at 23:44

If you're compiling a C-like language, say:

```
func()
{
    var a = 1;
    var *b = &a;
    *b = 2;
    assert(a == 2);
}
```

into Python, then all of the "everything in Python is a reference" stuff is a misnomer.

It's true that everything in Python is a reference, but the fact that many core types (ints, strings) are immutable effectively undoes this for many cases. There's no *direct* way to implement the above in Python.

Now, you can do it indirectly: for any immutable type, wrap it in a mutable type. Ephemient's solution works, but I often just do this:

```
a = [1]
b = a
b[0] = 2
assert a[0] == 2
```

(I've done this to work around Python's lack of "nonlocal" in 2.x a few times.)

This implies a lot more overhead: every immutable type (or every type, if you don't try to distinguish) suddenly creates a list (or another container object), so you're increasing the overhead for variables significantly. Individually, it's not a lot, but it'll add up when applied to a whole codebase.

You could reduce this by only wrapping immutable types, but then you'll need to keep track of which variables in the output are wrapped and which aren't, so you can access the value with "a" or "a[0]" appropriately. It'll probably get hairy.

As to whether this is a good idea or not--that depends on why you're doing it. If you just want something to run a VM, I'd tend to say no. If you want to be able to call to your existing language from Python, I'd suggest taking your existing VM and creating Python bindings for it, so you can access and call into it from Python.

answered Jul 17 '09 at 22:01
Glenn Maynard
37.6k 5 79 116

Yeah, my `ref` is basically the degenerate 1-element-only case of `list` . A 1- `tuple` would be lower overhead, but unfortunately those are not mutable. – ephemient Jul 17 '09 at 22:39

1 By the way, I think what you *really* want is a cell--that's what Python uses to store closures. Unfortunately, those are an implementation detail that aren't exposed--you can't instantiate them directly. – Glenn Maynard Jul 17 '09 at 22:55

Almost exactly like ephemient answer, which I voted up, you could use Python's builtin property function. It will do something nearly similar to the `ref` class in ephemient's answer, except now, instead of being forced to use `get` and `set` methods to access a `ref` instance, you just call the attributes of your instance which you've assigned as *properties* in the class definition. From Python docs (except I changed **C** to **ptr**):

```
class ptr(object):
    def __init__(self):
        self._x = None
    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

Both methods work like a C pointer, without resorting to `global` . For example if you have a function that takes a pointer:

```
def do_stuff_with_pointer(pointer, property, value):
    setattr(pointer, property, value)
```

For example

```
a_ref = ptr()      # make pointer
a_ref.x = [1, 2]   # a_ref pointer has an array [1, 2]
b_ref = a_ref      # b_ref points to a_ref
# pass ``ptr`` instance to function that changes its content
do_stuff_with_pointer(b_ref, 'x', 3)
print a_ref.x      # outputs 3
print b_ref.x      # outputs 3
```

Another, and totally crazy option would be to use Python's ctypes. Try this:

```
from ctypes import *
a = py_object([1,2]) # a has an array
b = a                # b points to a
b.value = 2          # derefernce b to store 2 in a
print a.value        # outputs 2
print b.value        # outputs 2
```

or if you want to get really fancy

```
from ctypes import *
a = py_object([1,2])   # a has an array
b = pointer(a)         # b points to a
b.contents.value = 2   # derefernce b to store 2 in a
print a.value          # outputs 2
print b.contents.value # outputs 2
```

which is more like OP's original request. crazy!

edited May 23 at 12:18
Community ♦
1 1

answered Dec 22 '12 at 7:32
Mark Mikofski
10.4k 1 30 56

Everything in Python is pointers already, but it's called "references" in Python. This is the translation of your code to Python:

```
a = [1,2]  // a has an array
b = a     // b points to a
a = 2      // store 2 in a.
print(a)   // outputs 2
print(b)  // outputs [1,2]
```

"Dereferencing" makes no sense, as it's *all* references. There isn't anything else, so nothing to dereference to.

answered Jul 17 '09 at 21:24
Lennart Regebro
88.4k 24 159 212

Thanks for the answer. What is concerning me is the last line of your answer. For the correct semantics in my old language I'd need be to now print out 2 as in the old language it's a pointer to a. – chollida Jul 17 '09 at 21:40

1 That's not a pointer then, but an alias, where you point one name to another name. You can't do that in Python, as that's completely unnecessary. In Python you would simply call it "a" the whole time. No aliases needed. – Lennart Regebro Jul 17 '09 at 21:51

As others here have said, all Python variables are essentially pointers.

The key to understanding this from a C perspective is to use the unknown by many id() function. It tells you what address the variable points to.

```
>>> a = [1,2]
>>> id(a)
28354600

>>> b = a
>>> id(a)
28354600

>>> id(b)
28354600
```

answered Jul 17 '09 at 21:43
Unknown
31.9k 18 113 165

3 The problem is how to dereference such an address. – brannerchinese Oct 17 '11 at 16:47

1 it's still very useful for identifying objects. without this, you'd need to hash the content or explicitly assign an id. in C, pointers are sometimes used to distinguish objects – UXkQEZ7 May 12 '14 at 5:22

"CPython implementation detail: This is the address of the object in memory." docs.python.org/2/library/functions.html#id – UXkQEZ7 May 12 '14 at 7:57

This is goofy, but a thought...

```
# Change operations like:
b = &a

# To:
b = "a"

# And change operations like:
*b = 2

# To:
locals()[b] = 2


>>> a = [1,2]
>>> b = "a"
>>> locals()[b] = 2
>>> print(a)
2
>>> print(locals()[b])
2
```

But there would be no pointer arithmetic or such, and no telling what other problems you might run into...

edited Jul 17 '09 at 21:59 answered Jul 17 '09 at 21:53
Anon

6,368 2 17 18

This probably isn't the greatest idea, because changes to locals() aren't guaranteed to be reflected in the environment. – Paul Fisher Jul 17 '09 at 22:00

Also, you can't pass `b` into other functions this way. – ephemient Jul 17 '09 at 22:37

As per Paul's comment - I tested it in a function, and it didn't even change the variable in there. Can change globals that way in a function, but locals() apparently is just giving a copy of the dict when in a function. – Anon Jul 17 '09 at 22:46

And still: Why? If you want to reference a, just use a. This is still just trying to make something that is easy in Python, but complicated in C, the complicated C way. – Lennart Regebro Jul 17 '09 at 22:54

@Lennart - I don't think anybody would argue its the way to write Python code. But it seems like what the OP was after was a simple mechanistic way to convert his code to working Python. And if that is the priority, then maybe finding a way to map pointers and dereferencing onto Python might achieve that goal, admittedly at the cost of resulting hideous - but working - Python code. – Anon Jul 17 '09 at 23:34

Negative, no pointers. You should not need them with the way the language is designed. However, I heard a nasty rumor that you could use the: ctypes module to use them. I haven't used it, but it smells messy to me.

answered Jul 17 '09 at 21:17

Alex
4,128 8 39 67

I know that there are no pointers in Python:) My goal is to cross compile my existing language into Python so I don't have to support a runtime any longer. What Python would I generate to simulate my example above? – chollida Jul 17 '09 at 21:21

```
class Pointer(object):
    def __init__(self, target=None):
        self.target = target

    _noarg = object()

    def __call__(self, target=_noarg):
        if target is not self._noarg:
            self.target = target
        return self.target
```

```
a = Pointer([1, 2])
b = a

print a() # => [1, 2]
print b() # => [1, 2]

b(2)
print a()  # => 2
print b()  # => 2
```

answered Nov 18 '14 at 8:28

community wiki
Jeremy Banks

I think that this example is short and clear.

Here we have class with implicit list:

```
class A:
   foo = []
a, b = A(), A()
a.foo.append(5)
b.foo
ans: [5]
```

Looking at this memory profile (using: `from memory_profiler import profile`), my intuition tells me that this may somehow simulate pointers like in C:

```
Filename: F:/MegaSync/Desktop/python_simulate_pointer_with_class.py

Line #    Mem usage    Increment   Line Contents
================================================
     7     31.2 MiB      0.0 MiB   @profile
     8                             def f():
     9     31.2 MiB      0.0 MiB       a, b = A(), A()
    10                                 #here memoery increase and is coupled
    11     50.3 MiB     19.1 MiB       a.foo.append(np.arange(5000000))
    12     73.2 MiB     22.9 MiB       b.foo.append(np.arange(6000000))
    13     73.2 MiB      0.0 MiB       return a,b


[array([      0,       1,       2, ..., 4999997, 4999998, 4999999]), array([
0,       1,       2, ..., 5999997, 5999998, 5999999])] [array([      0,       1,
2, ..., 4999997, 4999998, 4999999]), array([      0,       1,       2, ...,
5999997, 5999998, 5999999])]
Filename: F:/MegaSync/Desktop/python_simulate_pointer_with_class.py

Line #    Mem usage    Increment   Line Contents
================================================
    14     73.4 MiB      0.0 MiB   @profile
    15                             def g():
    16                                 #clearing b.foo list clears a.foo
    17     31.5 MiB    -42.0 MiB       b.foo.clear()
    18     31.5 MiB      0.0 MiB       return a,b


[] []
Filename: F:/MegaSync/Desktop/python_simulate_pointer_with_class.py

Line #    Mem usage    Increment   Line Contents
================================================
    19     31.5 MiB      0.0 MiB   @profile
    20                             def h():
    21                                 #and here mem. coupling is lost ;/
    22     69.6 MiB     38.1 MiB       b.foo=np.arange(10000000)
    23                                 #memory inc. when b.foo is replaced
    24    107.8 MiB     38.1 MiB       a.foo.append(np.arange(10000000))
    25                                 #so its seams that modyfing items of
    26                                 #existing object of variable a.foo,
    27                                 #changes automaticcly items of b.foo
    28                                 #and vice versa,but changing object
    29                                 #a.foo itself splits with b.foo
    30    107.8 MiB      0.0 MiB       return b,a


[array([      0,       1,       2, ..., 9999997, 9999998, 9999999])] [      0
1       2 ..., 9999997 9999998 9999999]
```

And here we have explicit self in class:

```
class A:
    def __init__(self):
        self.foo = []
a, b = A(), A()
a.foo.append(5)
b.foo
ans: []

Filename: F:/MegaSync/Desktop/python_simulate_pointer_with_class.py

Line #    Mem usage    Increment   Line Contents
================================================
    44    107.8 MiB      0.0 MiB   @profile
    45                             def f():
    46    107.8 MiB      0.0 MiB       a, b = B(), B()
    47                                 #here some memory increase
    48                                 #and this mem. is not coupled
    49    126.8 MiB     19.1 MiB       a.foo.append(np.arange(5000000))
    50    149.7 MiB     22.9 MiB       b.foo.append(np.arange(6000000))
    51    149.7 MiB      0.0 MiB       return a,b


[array([      0,       1,       2, ..., 5999997, 5999998, 5999999])] [array([
0,       1,       2, ..., 4999997, 4999998, 4999999])]
Filename: F:/MegaSync/Desktop/python_simulate_pointer_with_class.py

Line #    Mem usage    Increment   Line Contents
================================================
    52    111.6 MiB      0.0 MiB   @profile
    53                             def g():
    54                                 #clearing b.foo list
    55                                 #do not clear a.foo
    56     92.5 MiB    -19.1 MiB       b.foo.clear()
    57     92.5 MiB      0.0 MiB       return a,b


[] [array([      0,       1,       2, ..., 5999997, 5999998, 5999999])]
Filename: F:/MegaSync/Desktop/python_simulate_pointer_with_class.py

Line #    Mem usage    Increment   Line Contents
================================================
    58     92.5 MiB      0.0 MiB   @profile
    59                             def h():
    60                                 #and here memory increse again ;/
    61    107.8 MiB     15.3 MiB       b.foo=np.arange(10000000)
    62                                 #memory inc. when b.foo is replaced
    63    145.9 MiB     38.1 MiB       a.foo.append(np.arange(10000000))
    64    145.9 MiB      0.0 MiB       return b,a


[array([      0,       1,       2, ..., 9999997, 9999998, 9999999])] [      0
1       2 ..., 9999997 9999998 9999999]
```

ps: I'm self learning programming (started with Python) so please do not hate me if I'm wrong.
Its just mine intuition, that let me think that way, so do not hate me!

answered Jul 6 '15 at 9:52
user1839053