Pointers? In **My** Python?

It’s more likely than you think!

Eli Holderness / @eliholderness
Developer Advocate at Anvil
what pointers are

what the `id` of a Python object is

how CPython can tell when you’re done with an object
What is a pointer?
void *a_pointer;

int *a_pointer_to_an_int;

char *a_pointer_to_a_char;
Pointer aliasing

... also known as ‘wait, I didn’t change that variable, did I?’
>>> a = [“my”, “cool”, “list”]
>>> a
[“my”, “cool”, “list”]
>>> b = a
>>> b[1] = "awesome"
>>> b
["my", "awesome", "list"]

>>> a
["my", "awesome", "list"]
>>> c = a.copy()
>>> c[1] = "amazing"
>>> c
["my", "amazing", "list"]

>>> a
["my", "awesome", "list"]
I heard you like pointers...
>>> a = [['a', 'b'], ['A', 'B']]
>>> a
[['a', 'b'], ['A', 'B']]
```python
>>> b = a.copy()
>>> b[0].append("c")
>>> b[0]
["a", "b", "c"]

>>> a[0]
["a", "b", "c"]
```
What if it’s pointers all the way down?
>>> c = a.deepcopy()
>>> c[1].append("C")
>>> c[1]
["A", "B", "C"]

>>> a[1]
["A", "B"]
= makes a new pointer to the same object

copy makes a new object, and copies the pointers contained in the original

deepcopy makes a new object and copies the values, all the way down

https://docs.python.org/3/library/copy.html
Tuples behaving badly
A tuple \( a \) is immutable

\( a[0] \) must point to the same object during the lifetime of \( a \)

So what if \( a[0] \) is mutable?
>>> a = ([1, 2, 3], ["x", "y"])
>>> a[0]
[1, 2, 3]
```python
>>> a[0].append(4)
>>> a[0]
[1, 2, 3, 4]
```
>>> a[1] += ["z"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a[1]
[“x”, “y”, “z”]
>>> a[1] += [“z”]

+= mutates (+) then assigns (=)
Object IDs
id(x)

unique constant

... for the lifetime of x
Many Python implementations use the object’s **address in memory** as its id, but not all!

**CPython** uses the memory address  
**Skulpt** generates and caches a random number
>>> a = ["a", "list"]
>>> id(a)
140000359895536
>>> b = a
>>> id(a), id(b)
(140000359895536, 140000359895536)
>>> c = a.copy()

>>> id(a), id(c)
(140000359895536, 140000359764000)
When are two objects actually the same?
>>> a = ["a", "list"]
>>> b = a
>>> c = a.copy()
```python
>>> a == b
True
>>> a is b
True
>>> a == c
True
>>> a is c
False
```
is uses id

\[
a \text{ is } b \Leftrightarrow \\
id(a) = \text{id}(b)
\]
== uses __eq__

... so what is __eq__?
__str__
__repr__
__init__
The `__eq__` method defines the behaviour of `==` when applied to instances of its class.
```python
class MyClass:
    def __eq__(self, other):
        return self is other
```
class MyNamedClass:
    def __init__(self, name):
        self.name = name
    
    def __eq__(self, other):
        return True
>>> a = MyNamedClass("a")
>>> b = MyNamedClass("b")
>>> a.name == b.name
False
>>> a == b
True
class MyUniqueClass:
    def __eq__(self, other):
        return False
>>> a = MyUniqueClass()

>>> a == a

False
Object Lifetimes
called when an object is about to be removed from memory
class MyDelClass:
    def __init__(self, name):
        self.name = name

    def __del__(self):
        print(f"deleting {self.name}!")
Python frees memory in two ways:

Reference Counting

Garbage Collection
>>> dave = MyDelClass("Dave")
>>> del dave
Deleting Dave!
>>> alice = MyDelClass("Alice")
>>> also_alice = alice
>>> del alice

>>> del also_alice
Deleting Alice!
Cyclic references
```python
>>> jane = MyDelClass("Jane")
>>> bob = MyDelClass("Bob")
>>> bob.friend = jane
>>> jane.friend = bob

>>> del jane
>>> del bob
```
Reference counting isn’t sufficient
CPython’s Garbage Collector
detects cyclic isolates
calls their finalizers (\_\_del\_\_)*
breaks the cyclic references
>>> import gc
```python
>>> gc.is_tracked("a string")
False
```
>>> gc.is_tracked(["a", "list"])
True
>>> jane = MyDelClass("Jane")
>>> gc.is_tracked(jane)
True
The GC uses an object’s traversal method to access all its pointers.
>>> my_list = ["a", "list"]
>>> gc.get_referents(my_list)
[‘list’, ‘a’]
jane = MyDelClass("Jane")
bob = MyDelClass("Bob")
bob.friend = jane
jane.friend = bob
del jane
del bob
>>> import gc
>>> gc.collect()
Deleting Jane!
Deleting Bob!
4
Fun with finalizers
class MyBadDelClass:
    def __init__(self, name):
        self.name = name

    def __del__(self):
        global person
        person = self

        print(f"deleting {self.name}!")
>>> jane = MyBadDelClass("Jane")
>>> bob = MyBadDelClass("Bob")
>>> bob.friend = jane
>>> jane.friend = bob
>>> del jane
>>> del bob
>>> gc.collect()
Deleting Jane!
Deleting Bob!
0

>>> person
<_main__.MyBadDelClass object at 0x7ff8ce65dd30>
>>> person.name
'Bob'

>>> person.friend.name
'Jane'
>>> jane
NameError: name 'jane' is not defined
>>> del person
>>> gc.collect()
4
Pointers? In My Python?

It’s more likely than you think!

Eli Holderness / @eliholderness
Developer Advocate at Anvil