14a-is-and-equals

May 1, 2016

1 'is' and 'equals' operators

2 'a is b'

- returns true if a and b refer to the SAME object in the heap

In [1]:
```
# a's reference is copied to b, so a and b refer to the same object

a = [1,2,3]
b = a
a is b
```

Out[1]:
```
True
```

In [2]:
```
# an int and a list can't be the same object

b = 5
a is b
```

Out[2]:
```
False
```

In [3]:
```
# there are TWO different list [1,2,3] objects in the heap

a = [1,2,3]
b = [1,2,3]

a is b
```

Out[3]:
```
False
```

3 'a == b'

- returns true if a 'equals' b
- runs the __eq__ method on a

In [4]:
```
# a and b have 'same structure' but are different objects in the heap

a = [1, 2, [3, 4]]
b = [1, 2, [3, 4]]

a is b
```

Out[4]:
```
False
```

In [5]:
```
'''
a == b
runs a's list __eq__ method
this is because the two lists will be compared
recursively.  'a == b' here means

a & b are both the same type, 'list', they have the same length, and

a[0] == b[0] because 1 == 1
a[1] == b[1] because 2 == 2
a[2] == b[2] because [3,4 ] == [3, 4] because 3 == 3, 4 == 4
'''

a == b
```

Out[5]: True

In [6]:
```
class foo:
    def __init__(self, n):
        self.n = n

    def __eq__(self, x):
        if not isinstance(x, foo):
            return False
        return self.n == x.n
```

In [7]:
```
a = foo(3)
b = foo(3)
a is b
```

Out[7]: False

In [8]:
```
a == 3
```

Out[8]: False

In [9]:
```
a == b
```

Out[9]: True

In [10]:
```
# inherit from 'list' - only changing one method

class list2(list):
    def __eq__(self, x):
        if not isinstance(x, list):
            return False
        lens = len(self)
        lenx = len(x)
        # only check first two elements at most
        check = min(2, lens, lenx)
        for j in range(check):
            if not self[j] == x[j]:
                return False
        return True

a = list2('zap')
```

```
b = list2('zat')
c = list2('foo')

[a, b, c, a == b, a == c]
```

Out[10]: [['z', 'a', 'p'], ['z', 'a', 't'], ['f', 'o', 'o'], True, False]

4 interning objects

- if a new object is desired that would be == to an existing one, reuse the existing one instead of making a new one
- sometimes done solely for efficiency
- sometimes to make singletons

In [11]:
```
# small integers are interned, large ones are not
a = 1
b = 1
c = 123456
d = 123456

[a is b, c is d]
```

Out[11]: [True, False]

In [12]:
```
# there are TWO different list [1,2,3] objects in the heap,
# but the interned ints are the same

a = [1,2,3]
b = [1,2,3]

[a is b, a==b, a[0] is b[0], a[1] is b[1], a[2] is b[2]]
```

Out[12]: [False, True, True, True, True]

In [13]:
```
# trick for finding largest interned integer

for j in range(1000):
    s = str(j)
    if not int(s) is int(s):
        print(j)
        break
```

```
257
```

In [14]:
```
# reference counts for some ints

import sys

[[j, sys.getrefcount(j)] for j in range(-4,4)]
```

Out[14]:
```
[[-4, 11],
 [-3, 27],
 [-2, 48],
 [-1, 747],
 [0, 3099],
 [1, 2374],
 [2, 886],
 [3, 588]]
```

```
In [15]: # all strings are interned
         a = "foobarzap"
         b = "foobarzap"

         a is b
```

```
Out[15]: True
```

5 make interned version of foo

- use static 'factory' method
- 'factory pattern' is extremely common in OOP
- use class variable to hold existing instances

```
In [16]: class foo:
             # class var
             existing = dict()

             # static/class method - no 'self' argument
             def factory(n):
                 if n in foo.existing:
                     return foo.existing[n]
                 f = foo(n)
                 foo.existing[n] = f
                 return f

             def __init__(self, n):
                 self.n = n

             def __eq__(self, x):
                 if not isinstance(x, foo):
                     return False
                 return self.n == x.n
```

```
In [17]: f3 = foo.factory(3)
         f4 = foo.factory(4)
         f33 = foo.factory(3)
         [f3 is f4, f3 == f4, f3 is f33, f3 == f33]
```

```
Out[17]: [False, False, True, True]
```

```
In [18]: # Said ints bigger than 256 aren't interned, but
         # Python seems to intern ints in the same expression

         x = [123456, 123456]
         y = 123456

         [x[0] is y, x[0] is x[1]]
```

```
Out[18]: [False, True]
```