Lecture #12: Mutable Data

Announcement: "UPE, the L&S CS Honors Society, is hosting a resume workshop in preparation for the Career Fair next week, and they'll be going over some resume tips and doing one-on-one resume critiques. It's on Tuesday, Feb. 14th at 5pm in 380 Soda, so if you're interested stop by and bring your resume!"

Quick Feature Review

- By default, an assignment in Python (including = and `for...in`), binds a name in the current environment frame.
- But within any function, one may declare particular variables to be *nonlocal* or *global*:

```
>>> x0, y0 = 0, 0
>>> def f1(x1):
...     y1 = 0
...     def f2(x2):
...         nonlocal x1
...         global x0
...         x0, x1 = 1, 2
...         y0, y1 = 1, 2
...     f2(0)
...     print(x0, x1, y0, y1)
...
>>> f1(0)
1, 2, 0, 0
>>> print(x0, y0)
1, 0
```

Quick Feature Review (II)

- global marks names assigned to in the function as referring to variables in the global scope, not new local variables. These variables need not previously exist, and may not already by local in the function.
- Old feature of Python.
- nonlocal marks names assigned to in function as referring to variables in some enclosing function. These variables must previously exist, and may not be local.
- Introduced in version 3 and immediate predecessors.
- Neither declaration affects variables in nested functions:

```
>>>def f():
...     global x
...     def g():  x = 3  # Local x
...     g()
...     return x
>>> x = 0
>>> f()
0
```

More on Building Objects With State

- The term *state* applied to an object or system refers to the current information content of that object or system.
- Include values of attributes and, in the case of functions, the values of variables in the environment frames they link to.
- Some objects are *immutable,* e.g., integers, booleans, floats, strings, and tuples that contain only immutable objects. Their state does not vary over time, and so objects with identical state may be substituted freely.
- Other objects in Python are (at least partially) *mutable*, and substituting one object for another with identical state may not work as expected if you expect that both objects continue to have the same value.
- Have seen that we can build mutable objects from functions.

Object Identity Versus Equality

- The == operator is intended to test for *equality of content* or *equivalence of state*.
- Two separate objects can therefore be ==.
- Sometimes (*BUT NOT OFTEN*) we need to see if two expressions in fact denote the same object.
- For this purpose, Python uses the operators `is` and `is not` analogously to == and !=.

```
>>> x = 1000000
>>> x == x + 1 - 1
True
>>> x is x + 1 - 1
False
>>> x = 100
>>> x == x + 1 - 1
True
>>> x is x + 1 - 1
True
```

```
>>> (1,) == (1,)
True
>>> (1,) is (1,)
False
>>> () == ()
True
>>> () is ()
True
```

```
>>> "a"*100 == "a"*100
True
>>> "a"*100 is "a"*100
False
>>> "a"*10 is "a"*10
True
```

WHAT'S GOING ON??

Object Identity Usually Irrelevant for Immutable Data

- The examples where `is` and `==` differ can differ from Python implementation to Python implementation.
- Runtime implementor is free to choose whether two expressions of literals that produce equal (==) values do so by producing identical (is) objects.
- This freedom results from the fact that, once equal, immutable values continue to be indistinguishable under equality.
- Again, this is *Referential transparency.*
- So when we write

```
>>> x = (2, 3)
>>> L = (1, x)
```

  it doesn't matter whether we create a new copy of `x` to put into `L`, or use the same one.

- . . . *Unless* we use is (which is why we generally don't!).

Mutable Objects With Functions (continued)

- We've already seen counters. How about dice?

```
import time
def make_dice(sides = 6, seed = None):
    """A new 'sides'-sided die that yields numbers between 1 and
    'sides' when called.  'seed' controls the sequence of values.
    If 'seed' is defaulted, chooses a non-deterministic value."""
    if seed == None:
        seed = int(time.time() * 100000)
    a, c, m = 25214903917, 11, 2**48  # From Java
    def die():
        nonlocal seed
        seed = (a*seed + c) % m
        return seed % sides + 1
    return die
>>> d = make_dice(6, 10002)
>>> d()
6
>>> d()
5
```

Mutable Objects With More Behavior

- Suppose we want objects with more than one operation on them?
- We did that for immutable tuples in Lecture #7 (`make_rat`). Can use the same technique.
- Example: Bank Accounts. I want this behavior:

```
def make_account(balance):
    """A new bank account with initial balance ‘balance‘.
    If acc is an account returned by make_account, then
    acc(’balance’) returns the current balance.
    acc(’deposit’, d) deposits d cents into the account
    acc(’withdraw’, w) withdraws w cents from the account, if
       possible.
    ’deposit’ and ’withdraw’ return acc itself."""
    ??
```

Aside: Helpful Piece of Syntax

```
>>> def foo(x, *y):
...     print(x, y)
...
>>> foo(1)
1 ()
>>> foo(1,2,3)
1 (2,3)
```

Bank Accounts Implemented

```
def make_account(balance):
    """A new bank account with initial balance ‘balance‘.
    If acc is an account returned by make_account, then
    acc('balance') returns the current balance.
    acc('deposit', d) deposits d cents into the account
    acc('withdraw', w) withdraws w cents from the account, if
       possible.
    'deposit' and 'withdraw' return acc itself."""

    def acc(op, *opnds):
        nonlocal balance
        if op == 'balance' and len(opnds) == 0:
            return balance
        elif op == 'deposit' and len(opnds) == 1 and opnds[0] > 0:
            balance += opnds[0]
            return acc
        elif op == 'withdraw' and len(opnds) == 1 \
            and 0 <= opnds[0] <= balance:
            balance -= opnds[0]
            return acc
        else: raise ValueError()
    return acc
```

Truth: We Don't Usually Do It This Way!

- Usually, if we want an object with mutable state, we use one of Pythons mutable object types.
- We'll see soon how to create such types.
- For now, let's look at some standard ones.

Lists

- Lists are mutable tuples, syntactically distinguished by `[...]`.
- Unlike tuples, therefore, we can assign to elements:

```
>>> x = [1, 2, 3]
>>> x[1] = 0
>>> x
[1, 0, 3]
```

- And can also assign to slices:

```
>>> x = [1, 2, 3]
>>> x[1:2] = [6, 7, 8]  # Replace 2nd item
>>> x
[1, 6, 7, 8, 3]
>>> x[0:2] = []         # Remove first 2
>>> x
[7, 8, 3]
```

Object Identity Is Now Important

```
>>> x = [1, 2]
>>> y = [0, x]
>>> x[0] = 5
>>> y
[0, [5, 2]]
>>> x = []
>>> y
?
```

Object Identity Is Now Important (II)

```
>>> x = [1, 2]
>>> y = [0, x]
>>> x[0] = 5
>>> y
[0, [5, 2]]
>>> x = []
>>> y
[0, [5, 2]]   # Why doesn't y change?
```