Lecture #13: Dictionaries and Mutable Functions

Announcements

- Project #2 (Maps) to be released after lecture.
- Homework #3 released sometime this evening.

Dictionaries

- Lists are a kind of *mutable function:*
  - Mappings from non-negative integers to values (but syntax differs).
  - Value at a particular integer can be changed, as can the domain (set of valid integer arguments).
- Tuples, likewise, are a kind of immutable function.
- *Dictionaries*, in effect, generalize lists.

Dictionaries

- *Dictionaries* (type dict) are mutable mappings from one set of values (called *keys*) to another.
- Constructors:

```
>>> {}   # A new, empty dictionary
>>> { 'brian' : 29, 'erik': 27, 'zack': 18, 'dana': 25 }
{'brian': 29, 'erik': 27, 'dana': 25, 'zack': 18}
>>> L = ('aardvark', 'axolotl', 'gnu', 'hartebeest', 'wombat')
>>> successors = { L[i-1] : L[i] for i in range(1, len(L)) }
>>> successors
{'aardvark': 'axolotl', 'hartebeest': 'wombat',
 'axolotl': 'gnu', 'gnu': 'hartebeest'}
```

- Queries:

```
>>> len(successors)
4
>>> 'gnu' in successors
True
>>> 'wombat' in successors
False
```

Dictionary Selection and Mutation

- Selection and Mutation

```
>>> ages = { 'brian' : 29, 'erik': 27, 'zack': 18, 'dana': 25 }
>>> ages['erik']
27
>>> ages['paul']
...
KeyError: 'paul'
>>> ages.get('paul', "?")
'?'
```

- Mutation:

```
>>> ages['erik'] += 1; ages['john'] = 56
ages
{'brian': 29, 'john': 56, 'erik': 28, 'dana': 25, 'zack': 18}
```

Dictionary Keys

- Unlike sequences, ordering is not defined.
- Keys must typically have immutable types that contain only immutable data [can you guess why?] that have a `__hash__` method. Take CS61B to find out what's going on here.
- When converted into a sequence, get the sequence of keys:

```
>>> ages = { 'brian' : 29, 'erik': 27, 'zack': 18, 'dana': 25 }
>>> list(ages)
['brian', 'erik', 'dana', 'zack']
>>> for name in ages: print(ages[name], end=",")
29, 27, 25, 18,
```

A Dictionary Problem

```
def frequencies(L):
    """A dictionary giving, for each w in L, the number of times w
    appears in L.
    >>> frequencies(['the', 'name', 'of', 'the', 'name', 'of', 'the',
    ...              'song'])
    {'of': 2, 'the': 3, 'name': 2, 'song': 1}
    """
```

Using Only Keys

- Suppose that all we need are the keys (values are irrelevant):

```
def is_duplicate(L):
    """True iff L contains a duplicated item."""
    items = {}
    for x in L:
        if x in items: return True
        items[x] = True   # Or any value
    return False
def common_keys(D0, D1):
    """Return dictionary containing the keys common to D0 and D1."
    result = {}
    for x in D0:
         if x in D1: result[x] = True
    return result
```

- These dictionaries function as *sets* of values.

Assignment up until Now

- By default, an assignment in Python (including `=` and `for...in`), binds a name in the *current environment frame.*
- Not always what you want. E.g.,

```
# Number of errors encountered
error_count = 0
def process_something(x):
    if not good(x):
        error_count += 1   # Doesn't work
    ...
```

The attempt to increment and assign to `error_count` creates a new local (uninitialized) variable.

Non-local and Global Assignment

- Within any function, may declare particular variables to be *nonlocal* or *global*:

```
>>> x0, y0 = 0, 0
>>> def g1(x1, y1):
...     def g2():
...         nonlocal x1
...         global x0
...         x0, x1 = 1, 2
...         y0, y1 = 1, 2
...     g2()
...     print(x0, x1, y0, y1)
...
>>> g1(0, 0)
1, 2, 0, 0
>>> print(x0, y0)
1, 0
```

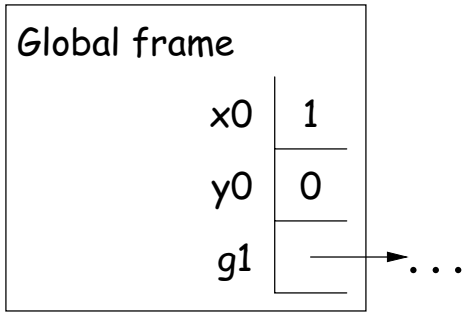
Details

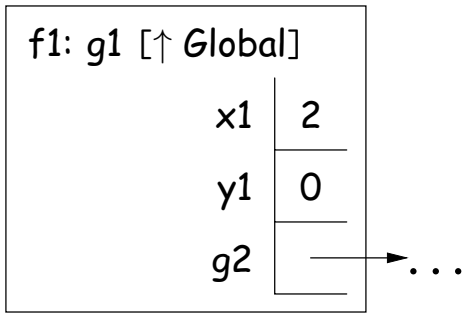
- `global` marks names assigned to in the function as referring to variables in the global scope, not new local variables. These variables need not previously exist, and must not already be local in the function.
- Old feature of Python.
- `nonlocal` marks names assigned to in function as referring to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

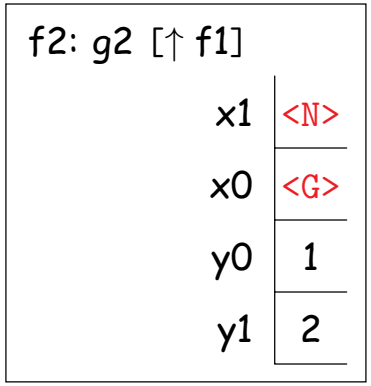
```
>>>def f():
...     global x
...     def g():  x = 3  # Local x
...     g()
...     return x
>>> x = 0
>>> f()
0
```

(Augmented) Environments for Non-Local Assignments

```
x0, y0 = 0, 0
def g1(x1, y1):
    def g2():
        nonlocal x1
        global x0
        x0, x1 = 1, 2
        y0, y1 = 1, 2
        # STOP HERE
    g2()
g1(0, 0)
```







- Not shown in Python tutor.
- The notation <G> means "look in the global frame for this variable."
- The notation <N> means "look in an ancestor frame (other than global) for this variable."
- Without these notations, assignments go in local frame as usual.

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.

Mutable Objects With Functions (continued)

- How about dice?

```
import time
def make_dice(sides = 6, seed = None):
    """A new 'sides'-sided die."""
    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."""
    ??
```

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."""

    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]
        elif op == ’withdraw’ and len(opnds) == 1 \
             and 0 <= opnds[0] <= balance:
             balance -= opnds[0]
        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.