Lecture #12: Mutable Data

Using Mutability For Construction: map_rlist Revisited

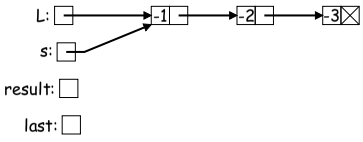
- Even if we never change a data structure once it is constructed, mutation may be useful *during* its construction.
- Example: constructing a recursive list. In lecture #9, I said that iterative construction of the result of `map_rlist` was not as easy as for `getitem_rlist`, compared to recursive version.
- But it's reasonably easy if we mutate items during construction:

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))
    s = rest(s)
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))
        last, s = rest(last), rest(s)
    return result
```


map_rlist Illustrated

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))
    s = rest(s)
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))
        last, s = rest(last), rest(s)
    return result
```

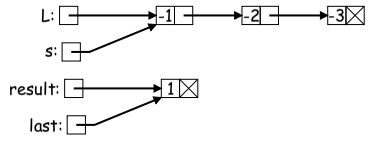
```
L = make_rlist(-1,
               make_rlist(-2,
                 make_rlist(-3)))
Q = map_rlist(abs, L)
```



map_rlist Illustrated (II)

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))    ⇐
    s = rest(s)
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))
        last, s = rest(last), rest(s)
    return result
```

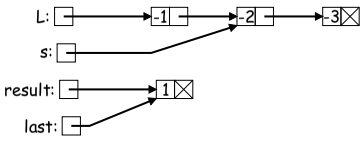
```
L = make_rlist(-1,
               make_rlist(-2,
                 make_rlist(-3)))
Q = map_rlist(abs, L)
```



map_rlist Illustrated (III)

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))
    s = rest(s)    ⇐
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))
        last, s = rest(last), rest(s)
    return result
```

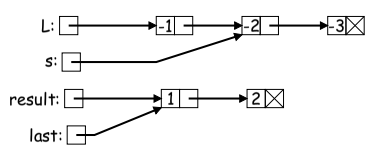
```
L = make_rlist(-1,
               make_rlist(-2,
                 make_rlist(-3)))
Q = map_rlist(abs, L)
```



map_rlist Illustrated (IV)

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))
    s = rest(s)
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))    ⇐
        last, s = rest(last), rest(s)
    return result
```

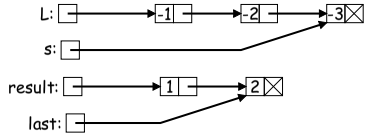
```
L = make_rlist(-1,
               make_rlist(-2,
                 make_rlist(-3)))
Q = map_rlist(abs, L)
```



map_rlist Illustrated (V)

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))
    s = rest(s)
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))
        last, s = rest(last), rest(s)    <=
    return result
```

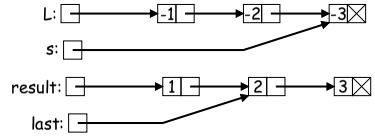
```
L = make_rlist(-1,
      make_rlist(-2,
        make_rlist(-3)))
Q = map_rlist(abs, L)
```



map_rlist Illustrated (VI)

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))
    s = rest(s)
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))    <=
        last, s = rest(last), rest(s)
    return result
```

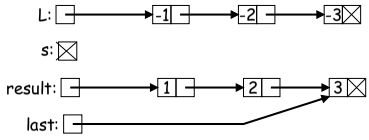
```
L = make_rlist(-1,
      make_rlist(-2,
        make_rlist(-3)))
Q = map_rlist(abs, L)
```



map_rlist Illustrated (VII)

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))
    s = rest(s)
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))
        last, s = rest(last), rest(s)    <=
    return result
```

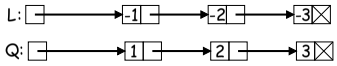
```
L = make_rlist(-1,
      make_rlist(-2,
        make_rlist(-3)))
Q = map_rlist(abs, L)
```



map_rlist Illustrated (VIII)

```
def map_rlist(f, s):
    """The rlist of values F(x) for each
    x in rlist S (in the same order.)"""
    if (isempty(s)):
        return s
    result = last = make_rlist(f(first(s)))
    s = rest(s)
    while not isempty(s):
        set_rest(last,
                 make_rlist(f(first(s))))
        last, s = rest(last), rest(s)
    return result
```

```
L = make_rlist(-1,
      make_rlist(-2,
        make_rlist(-3)))
Q = map_rlist(abs, L)    <=
```



- In building `Q`, we modified rlists we had previously created,...
- ...but `map_rlist` is non-destructive; the original list is intact.

Immutable and Mutable Data as Functions

- We've seen functions as immutable data items.
- For example, in lecture #8, we defined

```
def cons(left, right):
    return lambda which: left if which else right
def left(pair): return pair(True)
def right(pair): return pair(False)
```

- Can one do `set_left` and `set_right` with this representation?

Mutation By Assignment?

- Why not use assignment?

```
def cons(left, right):
    def data(which, value=None):
        if which == 0: return left
        elif which == 1: return right
        elif which == 2: left = value
        else: right = value
    return data
def left(pair): return pair(0)
def right(pair): return pair(1)
def set_left(pair, v): return pair(2, v)
def set_right(pair, v): return pair(3, v)
```

- This does not work. Why not?

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.,

```
def cons(left, right):
    def data(which, value=None):
        if which == 0: return left
        elif which == 1: return right
        elif which == 2: left = value  Doesn't work
        else: right = value            Doesn't work
    return data
A = cons(1, 2)
A(2, 4)   #  Try to assign 4 to left
```

- The attempt to assign to `left` creates a new local (uninitialized) variable on each call to `A`, which vanishes when the call returns.

The nonlocal Declaration

- To fix this problem, we introduce a new declaration: nonlocal:

```
def cons(left, right):
    def data(which, value=None):
        nonlocal left, right
        if which == 0: return left
        elif which == 1: return right
        elif which == 2: left = value   # Assigns to enclosing left
        else: right = value             # Assigns to enclosing right
    return data
A = cons(1, 2)
A(2, 4)   #  Try to assign 4 to left
```

- The effect of `nonlocal` is that all references `left` and `right` immediately within `data` refer to the ordinary local variable or parameter in the *smallest enclosing function definition,* rather than to any local variable in `data`.
- [Any `nonlocal` declarations in functions enclosing `data` would have no effect.]

Global Declaration

- `nonlocal` does not refer to *global variables*—those defined outside of any function.
- Instead, Python has a `global` declaration that marks names assigned in the function as referring to variables in the global scope.
- These variables need not previously exist, and must not already be local in the function.

```
>>> def f():
...     global x, y
...     x = 4            # Sets global x
...     y = 2            # Creates and sets global y
...     g()
>>> x = 1
>>> f()
>>> print(x, y)
4 2
```


Details

- Neither `global` nor `nonlocal` affects variables in more deeply nested functions:

```
>>>def f():
...     global x
...     def g():
...         x = 3  # Local x
...     g()
...     return x
>>> x = 0
>>> f()
0            # global declaration does not apply to outer x
```


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 incorrectly expect that both objects will continue to have the same value.
- Have just 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
```

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

- Usually, if we want an object with mutable state, we use one of Python's mutable object types,
- Let's look at a couple of standard ones.

Tuples and Lists

- Python tuples are a kind of function, mapping non-negative integers (indices) in a finite range to values.
- One cannot change the value at a given index, but can only create a new tuple:

```
>>> A = B = (1, 2, 3, 4, 5, 6)
>>> A[2] = 42; A[6:] = [7, 8]    # Illegal
>>> B = A[:2] + (42,) + A[3:] + (7, 8)
>>> A
(1, 2, 3, 4, 5, 6)
>>> B
(1, 2, 42, 4, 5, 6, 7, 8)
```

- Lists are a kind of *mutable function*, where the value at an index may be changed, and new items added.

```
>>> A = B = [1, 2, 3, 4, 5, 6]
>>> A[2] = 42;  A[6:] = [7, 8]
>>> A
[1, 2, 42, 4, 5, 6, 7, 8]
>>> B
[1, 2, 42, 4, 5, 6, 7, 8]
```


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', "?")  #  Supply default value
'?'
```

- 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}
    """
```

A Dictionary Problem (II)

```
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}
    """
    _____________________________

    for _____________________________:

        _____________________________

    return _____________________
```


A Dictionary Problem (III)

```
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}
    """
    result = {}

    for _____________________________:

        _____________________________

    return result
```


A Dictionary Problem (IV)

```
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}
    """
    result = {}

    for w in L:

        _____________________________

    return result
```


A Dictionary Problem (V)

```
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}
    """
    result = {}

    for w in L:

        result[w] = result.get(w, 0) + 1

    return result
```

Challenge: Do this in one line (I used 51 characters, including the `return`).

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.

Sets

Rather than force us to use dictionaries like this ("wasting" the values), Python supplies *sets*:

```
>>> rainbow = {'Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet'}
>>> nothing = set()  #  Empty set (sorry; {} was already taken)
>>> from_list = set([1, 2, 3])   #  Same as { 1, 2, 3 }
>>> A = { -2, -1, 0, 1, 2, 3, 4, 5 }
>>> B = { 0, 2, 4, 6, 8 }
>>> A.add(-3)   #  Mutable
>>> A | B       #  Union
{0, 1, 2, 3, 4, 5, 6, 8, -2, -3, -1}   #  Order undefined
>>> A & B       #  Intersection
{0, 2, 4}
>>> A - B       #  Set difference
{1, 3, 5, -1, -3, -2}
>>> A ^ B       #  Symmetric difference
{1, 3, 5, 6, 8, -1, -3, -2}
>>> 1 in B      #  Membership ( 1 ∈ B )
False
>>> A |= { 42 } #  Updating assignment (also \&, -, etc.)
>>> A
{0, 1, 2, 3, 4, 5, 42, -2, -3, -1}
```

Using Sets

- Can improve on previous use of dictionaries:

```
def is_duplicate(L):
    """True iff L contains a duplicated item."""
    return len(L) != len(set(L))
def common_keys(D0, D1):
    """Return set containing the keys common to D0 and D1."""
    return D0.keys() & D1.keys()
```

- When a dictionary is iterated over in a for loop, or turned into a list or set, the values it provides are its keys, so we can write the last line above as

```
    return set(D0) & set(D1)
```