UC Berkeley EECS
Lecturer Michael Ball

Computational Structures in Data Science

Lecture 12: Mutability

March 9, 2020

http://inst.eecs.berkeley.edu/~cs88

Announcements

- Maps project due Wed 4/1
- Midterm scores out tomorrow
- Watch Piazza for announcements about labs and office hours
- We will not be tracking participation today, but hope you still check in

Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
- Dictionaries
- Data structures
- Tuple assignment
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
  - Linear, Tail, Tree
- **Abstract Data Types**: Mutability

Review: Creating an Abtract Data Type

- **Operations**
  - Express the behavior of objects, invariants, etc
  - Implemented (abstractly) in terms of Constructors and Selectors for the object
- **Representation**
  - Constructors & Selectors
  - Implement the structure of the object

- An *abstraction barrier violation* occurs when a part of the program that can use the higher level functions uses lower level ones instead
  - At either layer of abstraction
- Abstraction barriers make programs easier to get right, maintain, and modify
  - Few changes when representation changes

Dictionaries – by example

- **Constructors:**
  - `dict( hi=32, lo=17)`
  - `dict([('hi',212),('lo',32),(17,3)])`
  - `{'x':1, 'y':2, 3:4}`
  - `{wd:len(wd) for wd in "The quick brown fox".split()}`
- **Selectors:**
  - **`water['lo']`**
  - **`<dict>.keys(), .items(), .values()`**
  - **`<dict>.get(key [, default] )`**
- **Operations:**
  - **`in, not in, len, min, max`**
  - `'lo' in water`
- **Mutators**
  - `water[ 'lo' ] = 33`

Objects

- An Abstract Data Type consist of data and behavior bundled together to abstract a view on the data
- An object is a concrete instance of an abstract data type.
- Objects can have state
  - mutable vs immutable
- Next lectures: Object-oriented programming
  - A methodology for organizing large(er) programs
  - A core component of the Python language
- In Python, every value is an object
  - All objects have attributes
  - Manipulation happens through method

Mutability

- **Immutable – the value of the object cannot be changed**
  - integers, floats, booleans
  - strings, tuples
- **Mutable – the value of the object can …**
  - Lists
  - Dictionaries

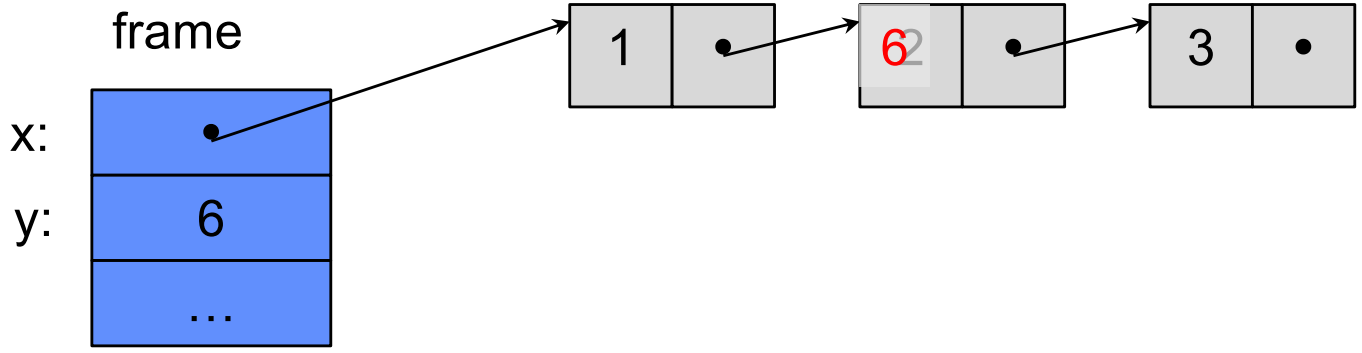
```
>>> alist = [1,2,3,4]
>>> alist
[1, 2, 3, 4]
>>> alist[2]
3
>>> alist[2] = 'elephant'
>>> alist
[1, 2, 'elephant', 4]
```

```
>>> adict = {'a':1, 'b':2}
>>> adict
{'b': 2, 'a': 1}
>>> adict['b']
2
>>> adict['b'] = 42
>>> adict['c'] = 'elephant'
>>> adict
{'b': 42, 'c': 'elephant', 'a':
1}
```

From value to storage …

- **A variable assigned a compound value (object) is a *reference* to that object.**
- **Mutable object can be changed but the variable(s) still refer to it**

```
x = [1, 2, 3]
y = 6
x[1] = y
x[1]
```



Mutation makes sharing visible

Python 3.6

```
1  x = 2
2  y = 3
3  print(x+y)
4  x = 4
5  print(x+y)
```

Edit this code

Print output (drag lower right corner to resize)

```
5
7
```

Frames Objects

Global frame

x	4
y	3

Python 3.6

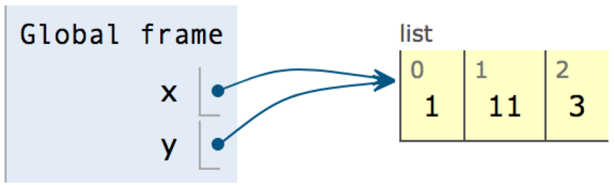
```
1  x = [1, 2, 3]
2  y = x
3  print(y)
4  x[1] = 11
5  print(y)
```

Edit this code

Print output (drag lower right corner to resize)

```
[1, 2, 3]
[1, 11, 3]
```

Frames Objects



Copies, 'is' and '=='

```
>>> alist = [1, 2, 3, 4]
>>> alist == [1, 2, 3, 4]  # Equal values?
True
>>> alist is [1, 2, 3, 4]  # same object?
False
>>> blist = alist          # assignment refers
>>> alist is blist         # to same object
True
>>> blist = list(alist)    # type constructors copy
>>> blist is alist
False
>>> blist = alist[ : ]     # so does slicing
>>> blist is alist
False
>>> blist
[1, 2, 3, 4]
>>>
```

Mutating Input Data

- Functions can mutate objects passed in as an argument
- Declaring a new variable with the same name as an argument only exists within the scope of our function
- BUT, we can still modify the object passed in, even though it was created in some other frame or environment.
- Python Tutor

Creating mutating 'functions'

- Pure functions have *referential transparency*
  - `c = greet() + name()` is "referentially transparent" if we can replace that expression with the value, maybe that's "Hello, CS 88"
- Result value depends only on the inputs
  - Same inputs, same result value
- Functions that use global variables are not pure
- They can be "mutating"

```
>>> counter = -1
>>> def count_fun():
...     global counter
...     counter += 1
...     return counter
...
>>> count_fun()
0
>>> count_fun()
1
```

Creating mutating 'functions'

```
>>> counter = -1
>>> def count_fun():
...     global counter
...     counter += 1
...     return counter
...
>>> count_fun()
0
>>> count_fun()
1
```

How do I make a second counter?

```
>>> def make_counter():
...     counter = -1
...     def counts():
...         nonlocal counter
...         counter +=1
...         return counter
...     return counts
...
>>> count_fun = make_counter()
>>> count_fun()
0
>>> count_fun()
1
>>> nother_one = make_counter()
>>> nother_one()
0
>>> count_fun()
2
```

Are these 'mutations' of seq?

```
def sum(seq):
    psum = 0
    for x in seq:
        psum = psum + x
    return psum

def reverse(seq):
    rev = []
    for x in seq:
        rev = [x] + rev
    return rev
```

A) Yes, both
B) Only sum
C) Only reverse
D) None of them

Solution:
D) No change of seq