Mutation

David E. Culler
CS8 – Computational Structures in Data Science
http://inst.eecs.berkeley.edu/~cs88
 Lecture 8
 October 15, 2018

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

C.O.R.E concepts
- Compute
- Operations
- Representation
- Evaluation

Abstract 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

Review: 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
- Objects represent information
- Consist of data and behavior, bundled together to create abstractions
  - Abstract Data Types
- They can have state
  - mutable vs immutable
- Object-oriented programming
  - A methodology for organizing large programs
  - So important it is supported in the language (classes)
- In Python, every value is an object
  - All objects have attributes
  - Manipulation happens through methods
- Functions do one thing (well)
  - Object do a collection of related things
Mutability

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

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

Are these ‘mutation’?

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

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

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

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

Mutation makes sharing visible

```python
Python 3.5
| y = 2 |
| y = 4 |
| y = 4 |
```

Examples

Sharing

Global frame

```
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
```

```python
grid:
```
```
Copies, ‘is’ and ‘==’

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

Creating mutating ‘functions’

- Pure functions have referential transparency
- Result value depends only on the inputs
- Functions that use global variables are not pure
- Higher order function returns embody state
- They can be “mutating”

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

Creating mutable objects

- Follow the ADT methodology, enclosing state within the abstraction

Useless bank account

```python
def account(name, initial_deposit):
    return (name, initial_deposit)
def account_name(acct):
    return acct[0]
def account_balance(acct):
    return acct[1]
def deposit(acct, amount):
    return (acct[0], acct[1]+amount)
def withdraw(acct, amount):
    return (acct[0], acct[1] - amount)

>>> my_acct = account('David Culler', 175)
>>> my_acct
('David Culler', 175)
>>> deposit(my_acct, 35)  # ('David Culler', 210)
>>> account_balance(my_acct)
175
```

Bank account using dict

```python
def account(name, initial_deposit):
    return {'Name': name, 'Number': 0, 'Balance': initial_deposit}
def account_name(acct):
    return acct['Name']
def account_balance(acct):
    return acct['Balance']
def deposit(acct, amount):
    acct['Balance'] += amount
    return acct['Balance']
def withdraw(acct, amount):
    acct['Balance'] -= amount
    return acct['Balance']

>>> your_acct = account('Fred Jones', 0)
>>> deposit(your_acct, 75)
75
>>> account_balance(my_acct)
175
```
State for a class of objects

```python
count = 0

def account(name, initial_deposit):
    global count
    count += 1
    return {'Name': name, 'Number': count, 'Balance': initial_deposit}
```

```python
>>> my_acct = account('David Culler', 100)
>>> my_acct
{'Name': 'David Culler', 'Number': 1, 'Balance': 100}
```

```python
Hiding the object inside

```python
count = 0

```python
def account(name, initial_deposit):
    global count
    count += 1
    return {'Name': name, 'Number': count, 'Balance': initial_deposit}
```

```python
>>> my_acct = account('David Culler', 100)
>>> my_acct
{'Name': 'David Culler', 'Number': 1, 'Balance': 100}
```

```python
A better way ...

```python
count = 0

```python
def account(name, initial_deposit):
    global count
    count += 1
    return {'Name': name, 'Number': count, 'Balance': initial_deposit}
```

```python
>>> my_acct = account('David Culler', 100)
>>> my_acct
{'Name': 'David Culler', 'Number': 1, 'Balance': 100}
```