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

Perform useful computations treating objects abstractly as whole values and operating on them.

Provide operations on the abstract components that allow ease of use – independent of concrete representation.

Constructors and selectors that provide an abstract interface to a concrete representation.

Execution on a computing machine.

Abstract Data Type

Abstraction Barrier

<table>
<thead>
<tr>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>adt operations</td>
</tr>
<tr>
<td>adt representation</td>
</tr>
</tbody>
</table>
Creating an 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]

>>> 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}
```
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
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.

\[
\begin{align*}
& x = [1, 2, 3] \\
& y = 6 \\
& x[1] = y \\
& x[1]
\end{align*}
\]
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

Global frame

x
y

list

0 1 11 3
Sharing

Global frame

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
>>> 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]
```
Creating mutating ‘functions’

• Pure functions have referential transparency
• Result value depends only on the inputs
  – Same inputs, same result value
• 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
...     return counter

>>> count_fun()
0
>>> count_fun()
1
```
Creating mutating ‘functions’

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

How do I make a second counter?
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
```
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']

>>> my_acct = account('David Culler', 93)
>>> account_balance(my_acct)
93
>>> deposit(my_acct, 100)
193
>>> account_balance(my_acct)
193
>>> withdraw(my_acct, 10)
183
>>> account_balance(my_acct)
183
>>> your_acct = account("Fred Jones",0)
>>> deposit(your_acct, 75)
75
>>> account_balance(your_acct)
75
State for a class of objects

```python
account_number_seed = 1000
def account(name, initial_deposit):
    global account_number_seed
    account_number_seed += 1
    return {'Name': name, 'Number': account_number_seed,
            'Balance': initial_deposit}

def account_name(acct):
    return acct['Name']

def account_balance(acct):
    return acct['Balance']

def account_number(acct):
    return acct['Number']

def deposit(acct, amount):
    acct['Balance'] += amount
    return acct['Balance']

def withdraw(acct, amount):
    acct['Balance'] -= amount
    return acct['Balance']

>>> my_acct = account('David Culler', 100)
>>> my_acct
{'Name': 'David Culler', 'Balance': 100,
 'Number': 1001}

>>> account_number(my_acct)
1001

>>> your_acct = account("Fred Jones", 475)
>>> account_number(your_acct)
1002
```
Hiding the object inside

```python
account_number_seed = 1000
accounts = []

def account(name, initial_deposit):
    global account_number_seed
    global accounts
    account_number_seed += 1
    new_account = {'Name': name, 'Number': account_number_seed,
                   'Balance': initial_deposit}
    accounts.append(new_account)
    return len(accounts)-1

def account_name(acct):
    return accounts[acct]['Name']

def deposit(acct, amount):
    account = accounts[acct]
    account['Balance'] += amount
    return account['Balance']

def account_by_number(number):
    for account, index in zip(accounts, range(len(accounts))):
        if account['Number'] == number:
            return index
    return -1
```
Hiding the object inside

```python
>>> my_acct = account('David Culler', 100)
0
>>> account_number(my_acct)
1001
>>> your_acct = account("Fred Jones", 475)
>>> accounts
[{'Name': 'David Culler', 'Balance': 100, 'Number': 1001},
 {'Name': 'Fred Jones', 'Balance': 475, 'Number': 1002}]
>>> account_by_number(1001)
0
>>> account_name(account_by_number(1001))
'David Culler'
>>> your_acct
1
>>> account_name(your_acct)
'Fred Jones'
```
def remove_account(acct):
    global accounts
    accounts = accounts[0:acct] + accounts[acct+1:]

>>> my_acct = account('David Culler', 100)
>>> your_acct = account("Fred Jones", 475)
>>> another_acct = account("Wilma Flintstone", 999)
>>> account_name(your_acct)
'Fred Jones'
>>> remove_account(my_acct)
>>> account_name(your_acct)
'Wilma Flintstone'
>>>
A better way ...

```python
account_number_seed = 1000
accounts = []

def account(name, initial_deposit):
    global account_number_seed
    global accounts
    account_number_seed += 1
    new_account = {'Name': name, 'Number': account_number_seed,
                   'Balance': initial_deposit}
    accounts.append(new_account)
    return account_number_seed

def _get_account(number):
    for account in accounts:
        if account['Number'] == number:
            return account
    return None

def account_name(acct):
    return _get_account(acct)['Name']
...
A better way ...

```python
account_number_seed = 1000
accounts = []

def account(name, initial_deposit):
    global account_number_seed
    global accounts
    account_number_seed += 1
    new_account = {'Name': name, 'Number': account_number_seed, 'Balance': initial_deposit}
    accounts.append(new_account)
    return account_number_seed

def _get_account(number):
    for account in accounts:
        if account['Number'] == number:
            return account
    return None

def account_name(acct):
    return _get_account(acct)['Name']

>>> my_acct = account('David Culler', 100)
>>> your_acct = account("Fred Jones", 475)
>>> nother_acct = account("Wilma Flintstone", 999)

>>> account_name(your_acct)
'Fred Jones'

>>> remove_account(my_acct)

>>> account_name(your_acct)
'Fred Jones'

>>> your_acct
1002

def _get_account(number):
    for account in accounts:
        if account['Number'] == number:
            return account
    return None

def account_name(acct):
    return _get_account(acct)['Name']

...