Computational Structures in Data Science

Lecture 8: Mutability

Review: 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

Dictionaries – by example

• Constructors:
  – `dict(hi=32, lo=17)`
  – `dict([('hi',212),('lo',32),(17,3)])`
  – `{x:1, y:2, z: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(ger) 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

```python
>>> alist = [1,2,3,4]
>>> alist
[1, 2, 3, 4]
>>> alist[2] = 'elephant'
>>> alist
[1, 2, 'elephant', 4]
>>> adict = {'a':1, 'b':2}
>>> adict
{'b': 2, 'a': 1}
>>> adict['b'] = 42
>>> adict
{'b': 42, 'a': 'elephant', '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
```

Mutant makes sharing visible

```
x = [1, 1, 1]
y = x
print(x)
```

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

Arguments are Mutable

• When you pass in list to a function, you can change the values of that function.
• True of lists and dictionaries, and other complex types.
• Not true of primitive types: integers, Booleans, strings, floats which are immutable.
• Python Tutor

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”

```
>>> counter = -1
>>> def count_fun():
...     global counter
...     counter += 1
...     return counter
...     how do i make a second counter?
```

Functions that Mutate

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

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

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

```
>>> def make_counter():
...     counter = -1
...     def count_fun():
...         global counter
...         counter += 1
...         return counter
...     return count_fun
...     make_count
```
END PART 1

Are these ‘mutations’?

```python
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`

Creating mutable objects

- Follow the ADT methodology, enclosing state within the abstraction

Useless bank account

```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']
```

>>> my_acct = account('David Culler', 175)
>>> my_acct
{'Name': 'David Culler', 'Balance': 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']
```

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

```python
>>> my_acct = account('David Culler', 175)
>>> my_acct
{'Name': 'David Culler', 'Number': 1001, 'Balance': 175}
>>> account_balance(my_acct)
175
```

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

```python
>>> my_acct = account('David Culler', 175)
>>> my_acct
{'Name': 'David Culler', 'Number': 1001, 'Balance': 175}
>>> account_balance(my_acct)
175
```
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

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)
>>> my_acct
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'
```

Hazard Beware

```python
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)
>>> nother_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']

>>> 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)
'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']

>>> 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)
'Wilma Flintstone'
```