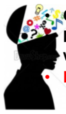
Computational Structures in Data Science

Lecture 8: Mutability

Oct 28, 2019 http://inst.eecs.berkeley.edu/~cs88

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

UCB CS88 Fa19 L08

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

UCB CS88 Fa19 L08

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`

UCB CS88 Fa19 L08

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

UCB CS88 Fa19 L08

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]
```

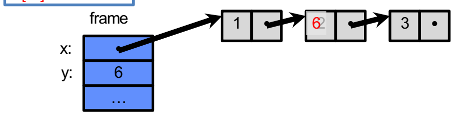
```
>>> adict = {'a':1, 'b':2}
>>> adict
{'b': 2, 'a': 1}
>>> adict['b']
2
>>> adict['b'] = 42
>>> adict['c'] = 'elephant'
>>> adict
{'b': 42, 'c': 'elephant', 'a':
1}
```

UCB CS88 Fa19 L08

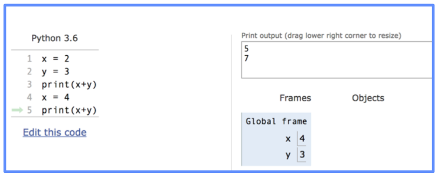
From value to storage …

- 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]
```



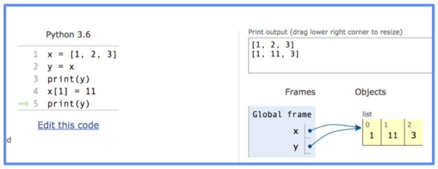
Mutation makes sharing visible



Python 3.6

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

```
5
7
```



Python 3.6

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

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

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]
>>>
```

Arguments are Mutable

- When you pass in lst 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
...
>>> count_fun()
0
>>> count_fun()
1
```

Functions that Mutate

```
>>> 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
```

END PART 1

Are these 'mutations' ?

```
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

```
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

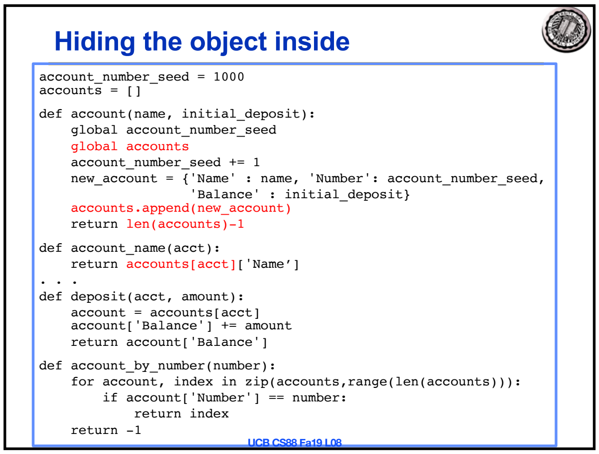
```
def account(name, initial_deposit):
    return {'Name' : name, 'Number': 0,
            'Balance' : initial_deposit}

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(my_acct)
183
```

State for a class of objects

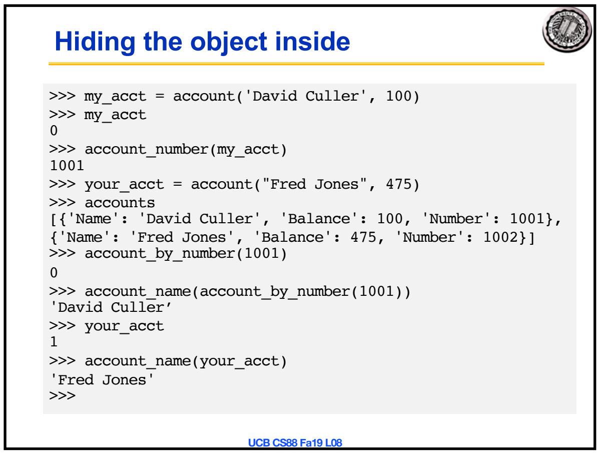
```
account_number_seed = 1000

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
>>>
```



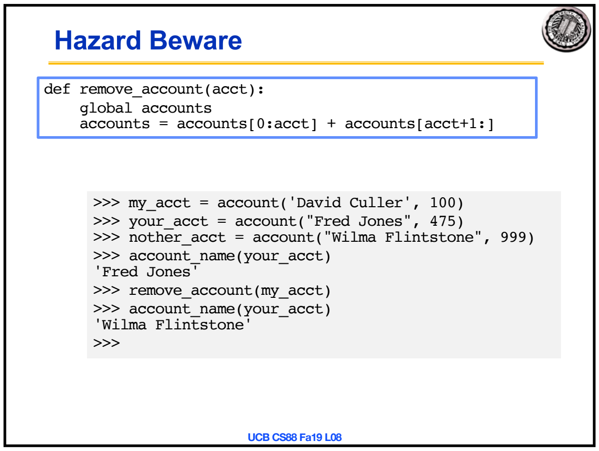
Hiding the object inside

```
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

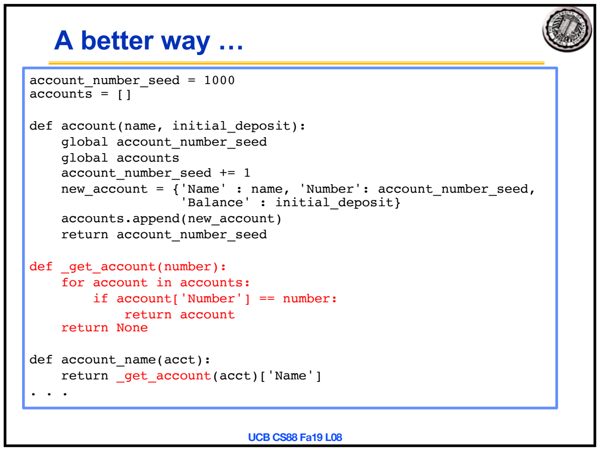
```
>>> my_acct = account('David Culler', 100)
>>> my_acct
0
>>> account_number(my_acct)
1001
>>> your_acct = account("Fred Jones", 475)
>>> accounts
[{'Name': 'David Culler', 'Balance': 100, 'Number': 1001},
{'Name': 'Fred Jones', 'Balance': 475, 'Number': 1002}]
>>> account_by_number(1001)
0
>>> account_name(account_by_number(1001))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



Hazard Beware

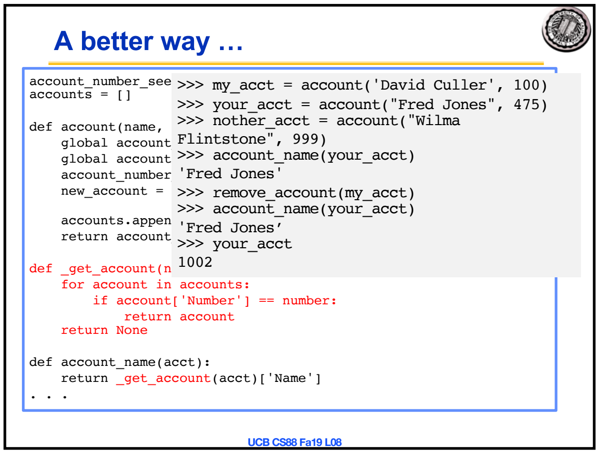
```
def remove_account(acct):
    global accounts
    accounts = accounts[0:acct] + accounts[acct+1:]
```

```
>>> 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 …

```
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 …

```
account_number_see
accounts = []

def account(name,
    global account
    global account
    account_number
    new_account =

    accounts.appen
    return account

def _get_account(n
    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
```