



UC Berkeley EECS
Lecturer Michael Ball

Computational Structures in Data Science



Lecture 11

Abstract Data Types



Computational Concepts Toolbox

- **Data type: values, literals, operations,**
 - e.g., int, float, string
- **Expressions, Call expression**
- **Variables**
- **Assignment Statement**
- **Sequences: tuple, list**
 - indexing
- **Data structures**
- **Tuple assignment**
- **Call Expressions**
- **Function Definition Statement**
- **Conditional Statement**
- **Iteration:**
 - data-driven (list comprehension)
 - control-driven (for statement)
 - while statement
- **Higher Order Functions**
 - Functions as Values
 - Functions with functions as argument
 - Assignment of function values
- **Recursion**
- **Lambda - function valued expressions**



Announcements

- **Midterm scores out early next week**
- **Please don't come to class if you're sick.**



Today's Lecture

- **Abstract Data Types**
 - More use of functions!
 - Value in documentation and clarity
- **New Python Data Types**
 - Dictionaries, a really useful tool!

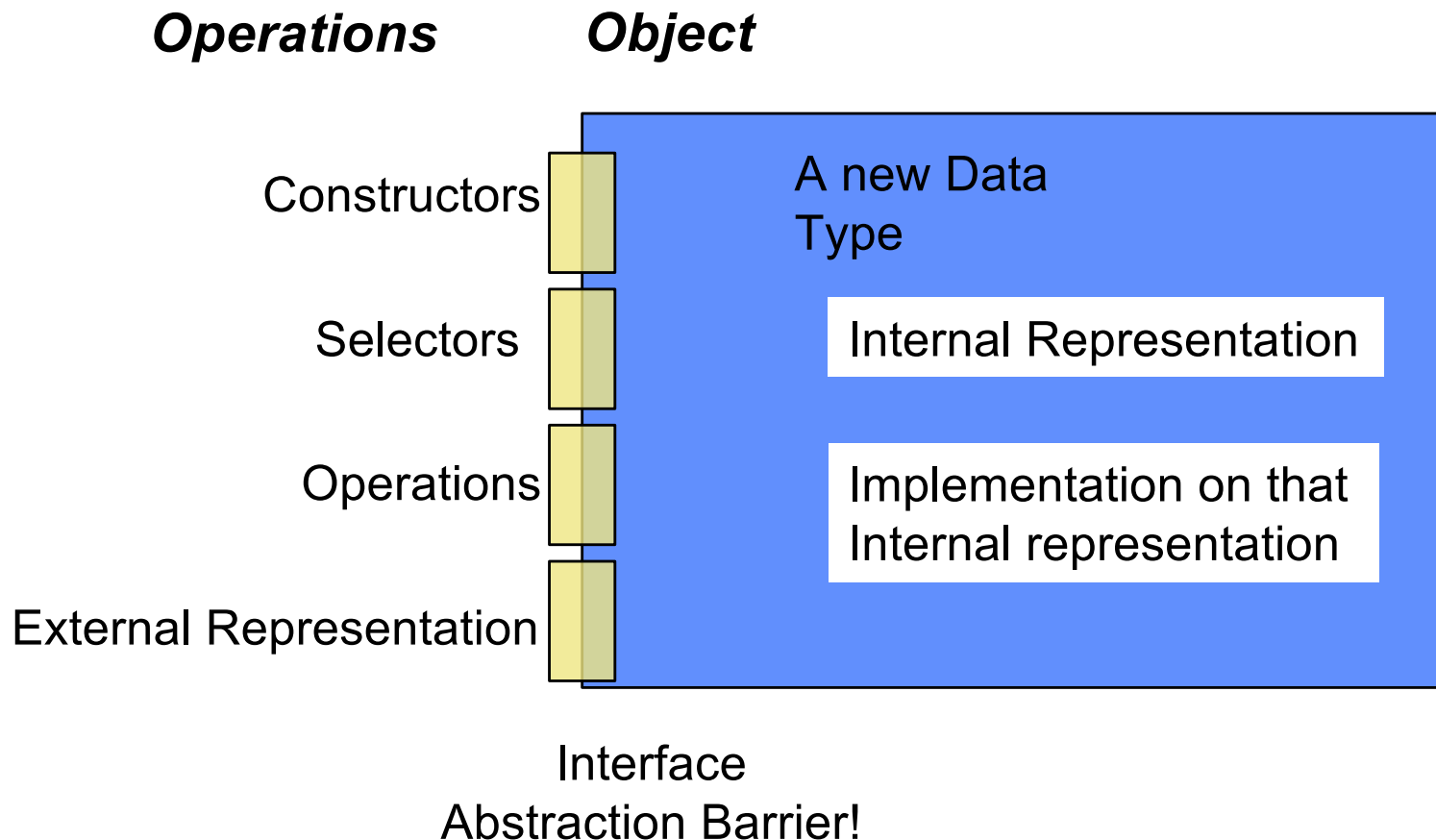


Why Abstract Data Types?

- **“Self-Documenting”**
 - `contact_name(contact)`
 - » Vs `contact[0]`
 - “0” may seem clear now, but what about in a week? 3 months?
- **Change your implementation**
 - Maybe today it’s just a Python List
 - Tomorrow: It could be a file on your computer; a database in web



Abstract Data Type





Creating Abstractions

- **Compound values combine other values together**
 - date: a year, a month, and a day
 - geographic position: latitude and longitude
 - a game board
- **Data abstraction lets us manipulate compound values as units**
- **Isolate two parts of any program that uses data:**
 - How data are represented (as parts)
 - How data are manipulated (as units)
- **Data abstraction: A methodology by which functions enforce an abstraction barrier between *representation* and *use***



Reminder: Lists

- **Lists**

- **Constructors:**

- » `list(...)`

- » `[<exprs>, ...]`

- » `[<exp> for <var> in <list> [if <exp>]]`

- **Selectors:** `<list> [<index or slice>]`

- **Operations:** `in`, `not in`, `+`, `*`, `len`, `min`, `max`

- » Mutable ones too (but not yet)

- » **Tuples**

- » A lot like lists, but you cannot edit them. We'll revisit on Monday.



A Small ADT

```
def point(x, y): # constructor
    return [x, y]
```

```
x = lambda point: point[0] # selector
y = lambda point: point[1]
```

```
def distance(p1, p2): # Operator
    return ((x(p2) - x(p1))**2 + (y(p2) -
y(p1))**2) ** 0.5
```

```
origin = point(0, 0)
my_house = point(5, 5)
campus = point(25, 25)
distance_to_campus = distance(my_house, campus)
```



Creating an Abstract Data Type

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



Clicker: Changing Representations?

Assuming we update our selectors, what are valid representations for our `point(x, y)` ADT?

Currently `point(1, 2)` is represented as `[1, 2]`

- A) `[y, x]` # `[2, 1]`
- B) `"X: " + str(x) + " Y: " + str(y)`
`"X: 1 Y: 2"`
- C) `str(x) + ' ' + str(y)` # `'1 2'`
- D) All of the above
- E) None of the above



A Layered Design Process

- **Build the application based entirely on the ADT interface**
 - Operations, Constructors and Selectors
- **Build the operations in ADT on Constructors and Selectors**
 - Not the implementation representation
 - This is the end of the abstraction barrier.
- **Build the constructors and selectors on some concrete representation**

Example: Tic Tac Toe and Phone Book



- See the companion notebook.



Clicker: Changing Representations?

Which of these *violates* a board ADT?

- A) `diag_left = diagonal(board, 0)`
- B) `board[0][2] = 'x'`
- C) `all_rows = rows(board)`
- D) `board = empty_board()`
- E) None of the above



A little application

```
phone_book_data = [  
    ("Christine Strauch", "510-842-9235"),  
    ("Frances Catal Bulloan", "932-567-3241"),  
    ("Jack Chow", "617-547-0923"),  
    ("Joy De Rosario", "310-912-6483"),  
    ("Casey Casem", "415-432-9292"),  
    ("Lydia Lu", "707-341-1254")  
]  
  
phone_book = pb_create(phone_book_data)  
  
print("Jack Chows's Number: ", pb_get(phone_book, "Jack Chow"))  
  
print("Area codes")  
area_codes = list(map(lambda x:x[0:3], pb_numbers(phone_book)))  
print(area_codes)
```



Dictionaries

- Lists, Tuples, Strings, Range
- Dictionaries
 - Constructors:
 - » `dict(<list of 2-tuples>)`
 - » `dict(<key>=<val>, ...)` # like kwargs
 - » `{ <key exp>:<val exp>, ... }`
 - » `{ <key>:<val> for <iteration expression> }`
 - Selectors: `<dict> [<key>]`
 - » `<dict>.keys(), .items(), .values()`
 - » `<dict>.get(key [, default])`
 - Operations:
 - » Key in, not in, len, min, max
 - » `<dict>[<key>] = <val>`





Dictionary Example

```
In [1]: text = "Once upon a time"
        d = {word : len(word) for word in text.split()}
        d
```

```
Out[1]: {'Once': 4, 'a': 1, 'time': 4, 'upon': 4}
```

```
In [2]: d['Once']
```

```
Out[2]: 4
```

```
In [3]: d.items()
```

```
Out[3]: [('a', 1), ('time', 4), ('upon', 4), ('Once', 4)]
```

```
In [4]: for (k,v) in d.items():
        print(k, "=>", v)
```

```
('a', '=>', 1)
('time', '=>', 4)
('upon', '=>', 4)
('Once', '=>', 4)
```

```
In [5]: d.keys()
```

```
Out[5]: ['a', 'time', 'upon', 'Once']
```

```
In [6]: d.values()
```

```
Out[6]: [1, 4, 4, 4]
```



Clicker ?: Dictionaries

- What is the result of the final expression?

```
my_dict = { 'course': 'CS 88', semester = 'Fall' }  
my_dict['semester'] = 'Spring'
```

```
my_dict['semester']
```

- A) 'Fall'
- B) 'Spring'
- C) Error



Limitations

- **Dictionaries are unordered collections of key-value pairs**
- Dictionary keys have two restrictions:
 - A key of a dictionary cannot be a list or a dictionary (or any *mutable type*)
 - Two keys cannot be equal; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value



Beware

- **Built-in data type `dict` relies on mutation**
 - Clobbers the object, rather than “functional” – creating a new one
- **Throws an errors of key is not present**
- **We will learn about mutation shortly**



Example 3

- KV represented as dict



C.O.R.E concepts

Abstract Data Type

Compute

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

Operations

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

Representation

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

Evaluation

Execution on a computing machine

Abstraction Barrier



Building Apps over KV ADT

```
friend_data = [  
    ("Christine Strauch", "Jack Chow"),  
    ("Christine Strauch", "Lydia Lu"),  
    ("Jack Chow", "Christine Strauch"),  
    ("Casey Casem", "Christine Strauch"),  
    ("Casey Casem", "Jack Chow"),  
    ("Casey Casem", "Frances Catal Bulloan"),  
    ("Casey Casem", "Joy De Rosario"),  
    ("Casey Casem", "Casey Casem"),  
    ("Frances Catal Bulloan", "Jack Chow"),  
    ("Jack Chow", "Frances Catal Bulloan"),  
    ("Joy De Rosario", "Lydia Lu"),  
    ("Joy De Lydia", "Jack Chow")  
]
```

- **Construct a table of the friend list for each person**