UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Abstract Data Types & Dictionaries

Today's Lecture

- Abstract Data Types
  - More use of functions!
  - Value in documentation and clarity

UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Dictionaries

Learning Objectives

- Dictionaries are a new type in Python
- Lists let us index a value by a number, or position.
- Dictionaries let us index data by other kinds of data.

Dictionaries vs Lists

- Lists: Collect any number of items together in a single entity
  - Each item has an *index* or position
  - A list has a distinct order.
- Dictionaries: Map a "key" to a "value"
  - Keys can be many things, but most commonly strings, numbers, tuple
  - Values can be any data
  - Items have no inherent order
- If we know an item's index, or a dictionary's key then looking up and item is fast and easy.

Dictionaries

- Constructors:
  - » `dict( <list of 2-tuples> )`
  - » `dict( <key>=<val>, ...) # like kwargs`
  - » `{ <key exp>:<val exp>, …  }`
  - » `{ <key>:<val> for <iteration expression> }`
    - `>>> {x:y for x,y in zip(["a","b"],[1,2])}`
    - `{'a': 1, 'b': 2}`
- 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]
```

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]
```

Abstract Data Type

- Uses pure functions to encapsulate some logic as part of a program.
- We rely of built-in types (int, str, list, etc) to build ADTs
- This is a contrast to object-oriented programming
  - Which is coming soon!

Creating Abstractions

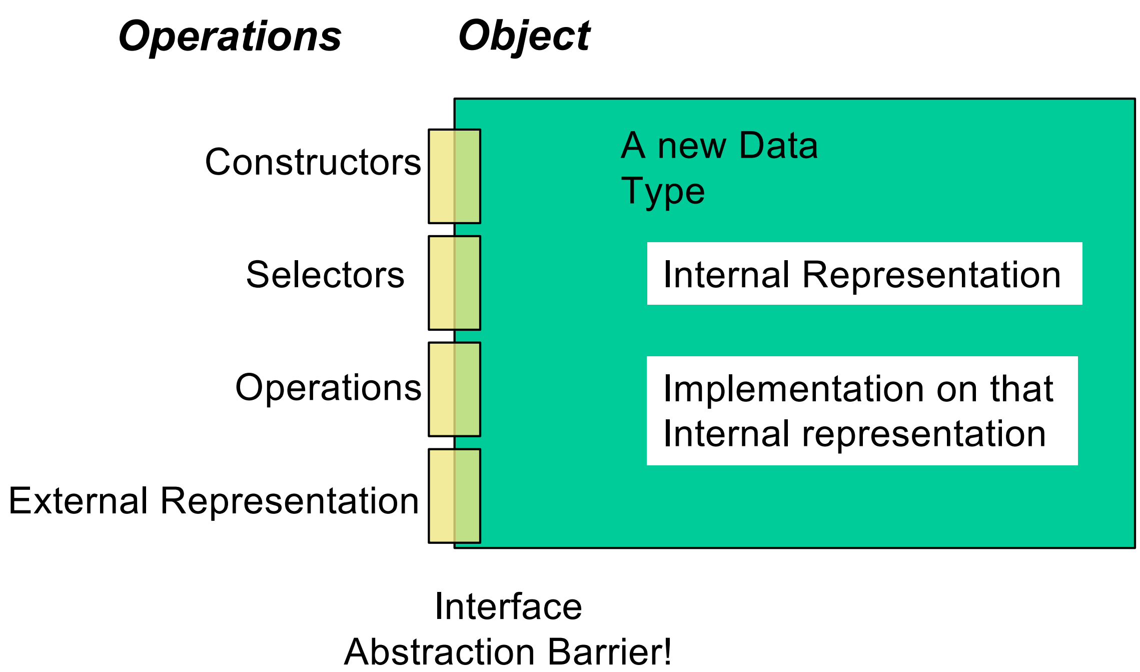
- Compound values combine other values together
  - date: a year, a month, and a day
  - geographic position: latitude and longitude
  - a game board

- 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*

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



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

Reminder: Lists

- Lists
  - Constructors:
    - list(...)
    - [<exps>,...]
    - [<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 Abtract 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

Question: Changing Representations?
http://go.c88c.org/9

Question 1.1

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.
- Download the file ″ipynb″
  - Go to datahub.berkeley.edu
  - Log in, then select ″Upload″

Question: The Abstraction Barrier

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