

Computational Structures in Data Science

Lecture: Dictionaries and Mutable Data



Computational Structures in Data Science

Environments Wrap Up



Announcements

- <https://go.c88c.org/8> - Self-Check
- <https://go.c88c.org/qa8> - Ed Thread

Bonus Self-Checks:

<https://www.gradescope.com/courses/1102730/assignments/6597540> - Lambdas

Python Tutor Example #2

```
def make_adder(n):  
    def adder(k):  
        return k + n  
    return adder  
  
n = 10  
add_2 = make_adder(2)  
x = add_2(5)
```

Python Tutor Example #3

```
add_2 = make_adder(2)
add_3 = make_adder(3)

x = add_2(2)
def compose(f, g):
    def h(x):
        return f(g(x))
    return h
add_5 = compose(add_2, add_3)
z = add_5(x)
```

Demo

Example 1:

- [make_adder Higher Order Function: Environment Diagram Python Tutor Link](#)

Example 2:

- [Primitives and Functions: Environment Diagram Python Tutor:](#)

Example 3:

- [Compose Python Tutor Link](#)

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

- 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>`

Demo

```
person = { 'name': 'Michael' }  
person.get('name')  
person['email'] = 'ball@berkeley.edu'  
person.keys()  
'phone' in person  
  
text = 'One upon a time'  
{ word : len(word) for word in text.split() }
```

Computational Structures in Data Science

Mutability

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Distinguish between when a function mutates data, or returns a new object
 - Many Python "default" functions return new objects
- Understand modifying objects in place
- Python provides "is" and "==" for checking if items are the same, in different ways

Why does Mutability Matter?

- Mutable data is a reality — lists, dictionaries, objects (coming soon)
- It's a challenging aspect of programming
- There are common patterns, which you will *slowly* become familiar with and internalize.
- Use your environment diagrams!

Objects in Python

- An **object** is a bundle of data and behavior.
- A type of object is called a **class**.
- Every value in Python is an object.
 - string, list, int, tuple, et
- All objects have attributes
- Objects often have associated methods
 - `lst.append()`, `lst.extend()`, etc
- **Objects have a value (or values)**
 - Mutable: We can change the object after it has been created
 - Immutable: We cannot change the object.
- Objects have an *identity*, a reference to that object.

Immutable Object: string

- `course = 'CS88'`
- What kind of object is it?
 - `type(course)`
- What data is inside it?
 - `course[0]`
 - `course[2:]`
- What methods can we call?
 - `course.upper()`
 - `course.lower()`
- None of these methods modify our original string.

Mutable Objects: lists and dictionaries

- Immutable – the value of the object cannot be changed
 - integers, floats, booleans
 - strings, tuples
- Mutable – the value of the object can change
 - 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}
```

Dictionaries

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
'name' in course
```

Mutators

```
course['number'] = 'C88C'
course.pop('room')
del course['room']
```

Immutability vs Mutability

- An immutable value is unchanging once created.
- Immutable types (that we've covered): int, float, string, tuple

```
a_string = "Hi y'all"  
a_string[1] = "I" # ERROR  
a_string += ", how you doing?"  
an_int = 20  
an_int += 2
```

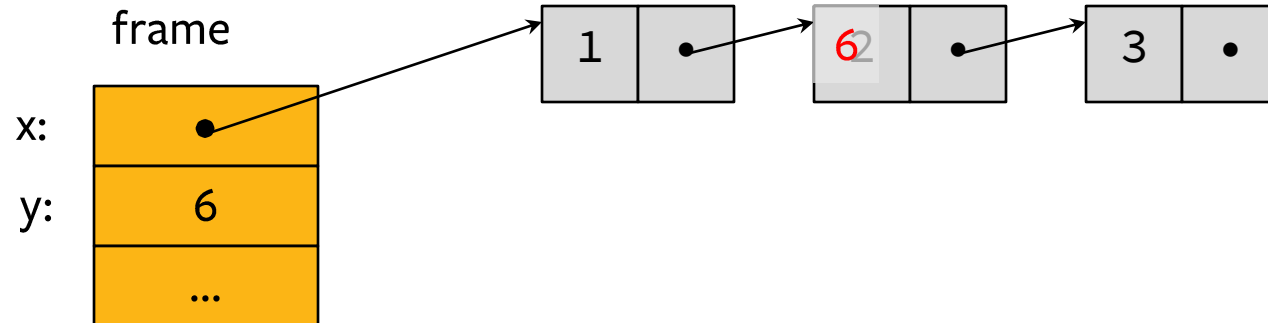
- A mutable value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.
- Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]  
grades_copy = grades # Not actually a copy!  
grades[1] = 100 # grades_copy changes too!  
words = {"agua": "water"}  
words["pavo"] = "turkey"
```

Mutation in Environments

- A variable assigned a compound value (object) is a reference to that object.
- Mutable objects can be changed but the variable(s) still refer to it
 - x is still the same object, but its values have changed.

```
x = [1, 2, 3]  
y = 6  
x[1] = y  
x[1]
```



Mutating Lists: Example functions of the `list` class

- `append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t.append(7)
t = 0
```

[Try in PythonTutor.](#)

- `extend()` adds all the elements in one list to another list:

```
s = [2, 3]
t = [5, 6]
s.extend(4) #  Error: 4 is not an iterable!
s.extend(t)
t = 0
```

[Try in PythonTutor.](#) (After deleting the bad line)

Mutating Lists -- More Functions!

- `list += [x, y, z]` # just like `extend`.
 - [You need to be careful with this one!](#) It modifies the list.
- `pop()` removes and returns the last element:

```
s = [2, 3]
```

```
t = [5, 6]
```

```
t = s.pop()
```

[Try in PythonTutor.](#)

- `remove()` removes the first element equal to the argument:

```
s = [6, 2, 4, 8, 4]
```

```
s.remove(4)
```

[Try in PythonTutor.](#)

Python Tutor: Assignments Are References

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	2
1	11	3

Mutable Data Inside Immutable Objects

- Mutable objects can "live" inside immutable objects!
- An immutable sequence may still change if it contains a mutable value as an element.
- Be **very careful**, and probably **do not** do this!

```
t = (1, [2, 3])
```

```
t[1][0] = 99
```

```
t[1][1] = "Problems"
```

- [Try in PythonTutor](#)

Equality vs Identity

```
list1 = [1,2,3]
```

```
list2 = [1,2,3]
```

- **Equality:** `exp0 == exp1`
evaluates to True if both `exp0` and `exp1` evaluate to objects containing equal values (Each object can define what `==` means)

```
list1 == list2 # True
```

- **Identity:** `exp0 is exp1`
evaluates to True if both `exp0` and `exp1` evaluate to the same object
- Identical objects always have equal values.

```
list1 is list2 # False
```

- [Try in PythonTutor.](#)

Identity and == vs is

How do we know if two names (variables) are the same exact object? i.e. Will modifying one modify the other?

```
>>> 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]
>>>
```

What is the meaning of `is`?

- `is` in Python means two items have the exact same *identity*
- Thus, `a is b` implies `a == b`
- **Why?** Each object has a function `id()` which returns its "address"
 - We won't get into what this means, but it's essentially an internal "locator" for that data in memory.
 - Think of two houses which have the exact same floor plan, look the same, etc. They are "the same house" but each has a unique address. (And thus are different houses)
- Think this is tricky? cool? amazing?
- Take CS61C (Architecture) and CS164 (Programming Languages)

Computational Structures in Data Science

Passing Data Into Functions



Learning Objectives

- Passing in a mutable object in a function in Python lets you modify that object
- Immutable objects don't change when passed in as an argument
- Making a new name doesn't affect the value outside the function
- Modifying mutable data **does** modify the values in the parent frame.

Mutating Arguments

- Functions can mutate objects passed in as an argument
- Declaring a new variable with the same name as an argument only exists within the scope of our function
 - You can think of this as creating a new name, in the same way as redefining a variable.
 - This will **not** modify the data outside the function, even for mutable objects.
- **BUT**
 - We can still directly modify the object passed in...even though it was created in some other frame or environment.
 - We directly call methods on that object.
- [View Python Tutor](#)

Understanding Python: What should we return?

- Why do some functions return **None**?
- Why do some functions return a value?

Functions that mutate an argument **usually** return None!

C88C / 61A / Data Science View: Avoid mutating data unless it's necessary!

Mutations are useful, but can get confusing quickly. This is why we focus on *functional programming* - map, filter, reduce, list comprehensions, etc.

Functions that Mutate vs Return New Objects

- Lists:
 - `sorted(list)` – returns a new list
 - `list.sort()` – modifies the list, returns `None`
 - `list.append()` – modifies the list, returns `None`
 - `list.extend()` – modifies the list, returns `None`

Python Gotcha's: `a += b` and `a = a + b`

- Sometimes similar **looking** operations have very different results!
- Why?
- `=` always binds (or re-binds) a value to a name.
- [Python Tutor](#)

```
def add_data_to_thing(thing, data):  
    print(f"+=, Before: {thing}")  
    thing += data  
    print(f"+=, After: {thing}")  
    return thing
```

```
def new_thing_with_data(thing, data):  
    print(f"=, Before: {thing}")  
    thing = thing + data  
    print(f"=, After: {thing}")  
    return thing
```

Computational Structures in Data Science

Mutable Functions



Learning Objectives

- Remember: Each function gets its own new frame
- Inner functions can access data in the parent environment
- Use an inner function along with a mutable data type to capture changes

Making Functions that Capture and change state

- We want to make a function, which returns a function that can change the state.
- [Python Tutor Link](#)

```
def make_counter():  
    counter = [0]  
    def count_up():  
        counter[0] += 1  
        return counter  
    return count_up
```

```
c = make_counter()  
print(c)  
c()  
c()  
c()
```

Functions with Changing State

- Goal: Use a function to repeatedly withdraw from a bank account that starts with \$100.
- Build our account: `withdraw = make_withdraw_account(100)`
- First call to the function:
`withdraw(25)` # 75
- Second call to the function:
`withdraw(25)` # 50
- Third call to the function:
`withdraw(60)` # 'Insufficient funds'

How Do We Implement Bank Accounts?

- A mutable value in the parent frame can maintain the local state for a function.
- [View in PythonTutor](#)

```
def make_withdraw_account(initial):  
    balance = [initial]  
  
    def withdraw(amount):  
        if balance[0] - amount < 0:  
            return 'Insufficient funds'  
        balance[0] -= amount  
        return balance[0]  
    return withdraw
```

Implementing Bank Accounts

- A mutable value in the parent frame can maintain the local state for a function.

```
def make_withdraw_account(initial):  
    balance = [initial]  
  
    def withdraw(amount):  
        if balance[0] - amount < 0:  
            return 'Insufficient funds'  
        balance[0] -= amount  
        return balance[0]  
    return withdraw
```

[View in PythonTutor](#)

Computational Structures in Data Science

Abstract Data Types



Abstract Data Type

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

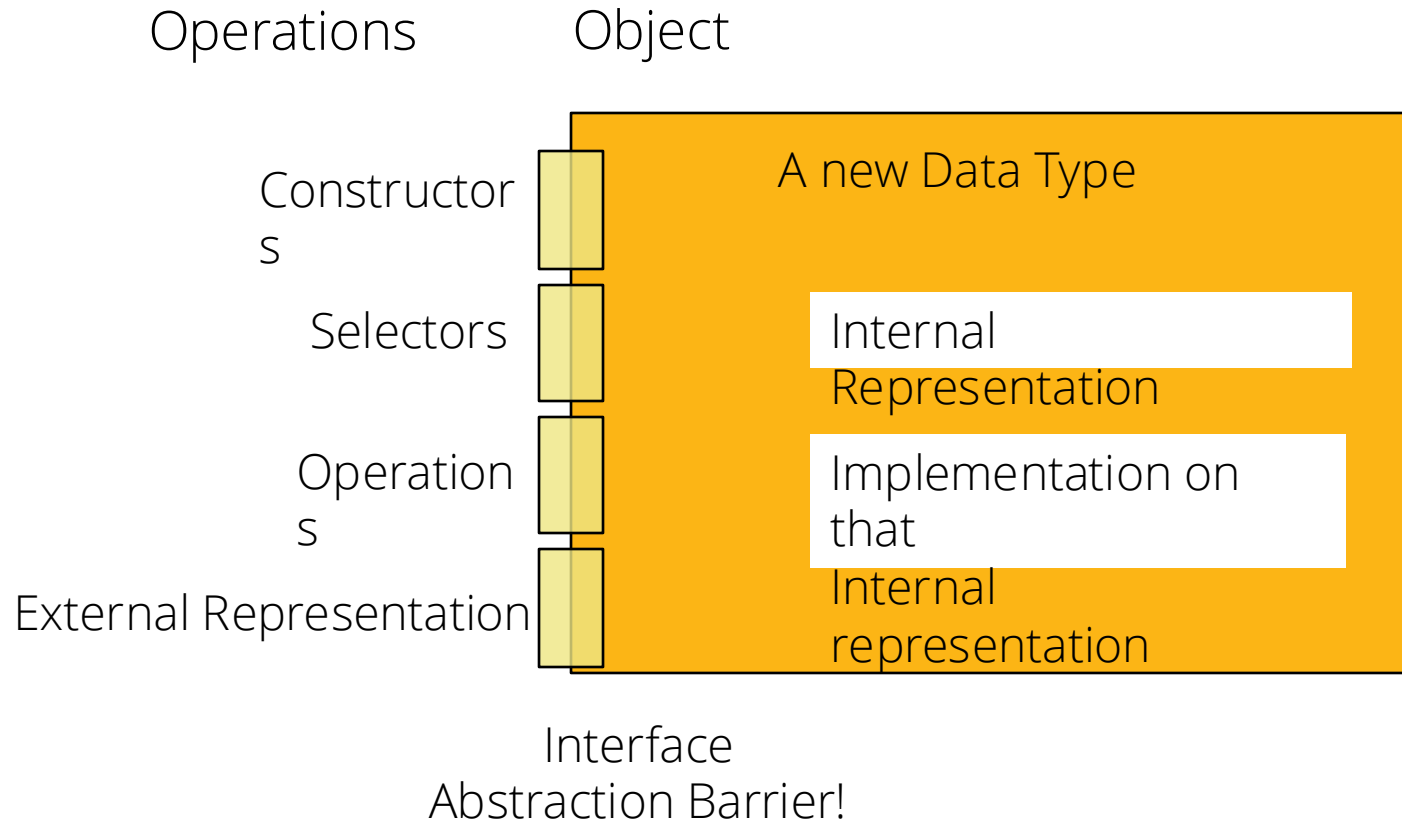
Creating Abstractions

- 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*

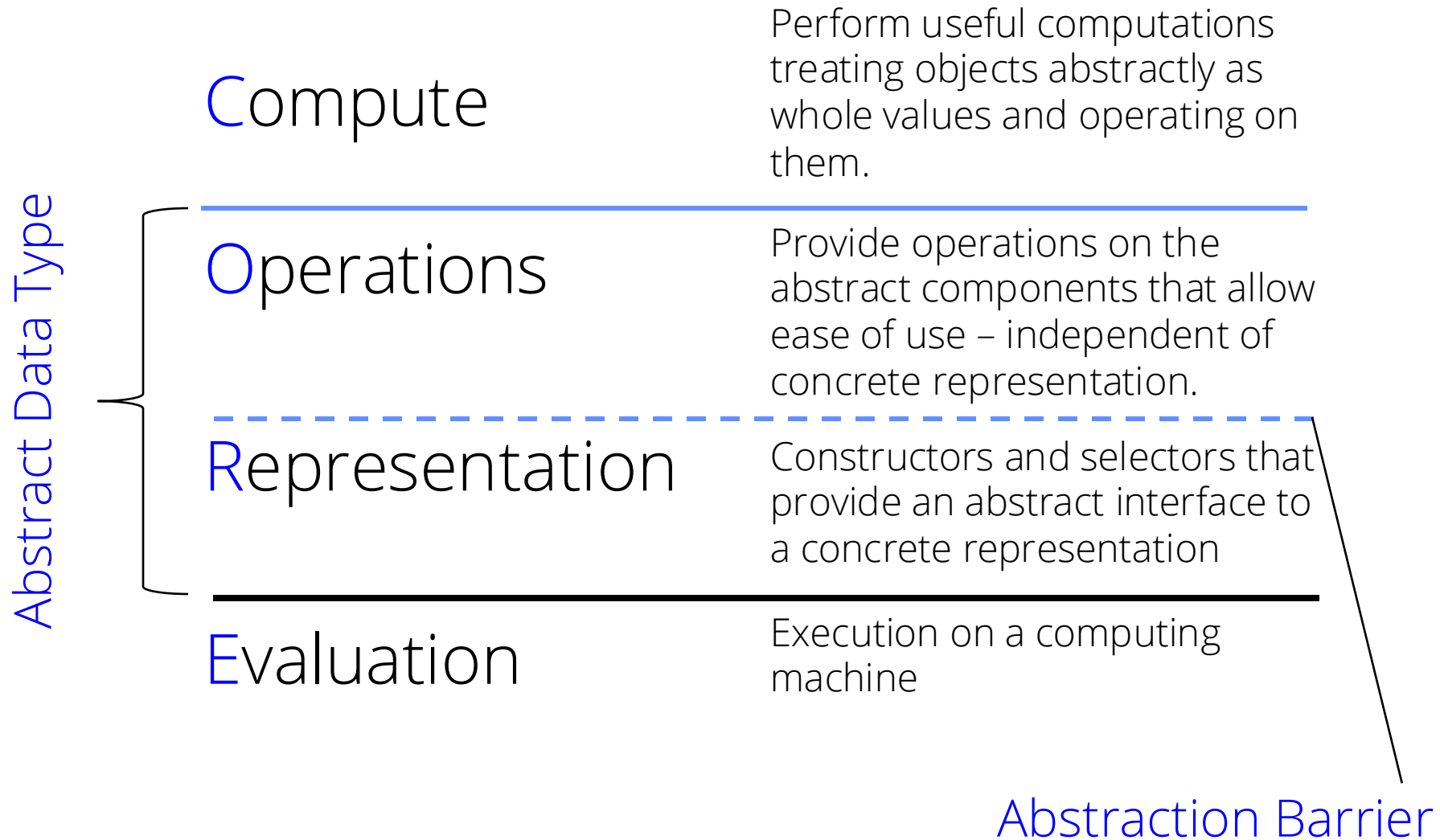
Why Abstract Data Types?

- How do you represent the *idea* of a game board, a "course", a person, a student, anything?
 - Programming languages allow you to do just about anything!
- "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



Reminder: Lists

- Lists
 - Constructors:
 - `list( ... )`
 - `[ <expr>, ... ]`
 - `[<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

Defining The Abstraction Barrier

- 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
 - e.g. Should your function be aware of the implementation?
 - Be consistent!
- Abstraction barriers make programs easier to get right, maintain, and modify
 - Fewer changes when representation changes

Question: Changing Representations? <http://go.c88c.org/8>

- 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 – Button Up

- Start with "What do you want to do?"
- Build the application based entirely on the ADT interface
 - Focus first on Operations, then Constructors and Selectors
 - Do not implement them! Your program won't work.
 - You want to capture the "user's" point of view
- 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

Optional 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"