Welcome to Data C88C!

Lecture 12: Objects
Thursday, July 10th, 2025
Week 3
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

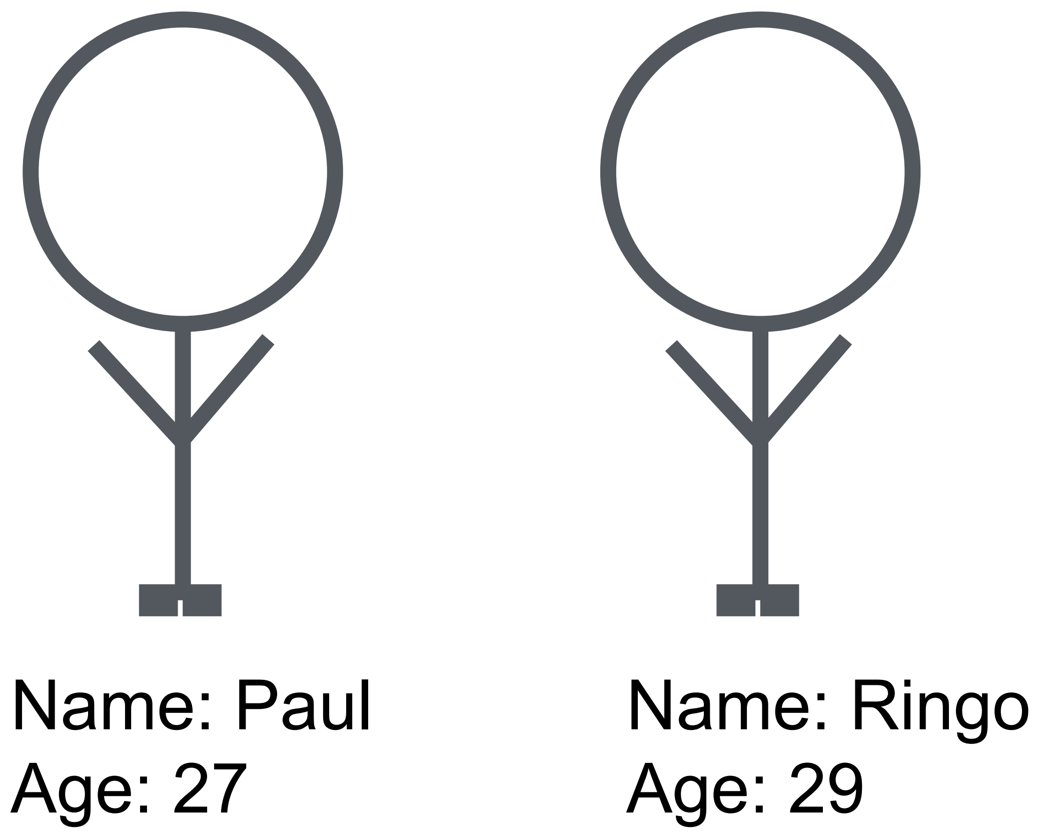
- Read midterm Ed post: [link]
- Lab07, HW07 released!
- Next week:
  - Midterm: Tuesday July 15th, 3pm - 5pm
  - No lecture Monday (July 14th), Tuesday (July 15th)
  - No labs (July 15th, July 17th)
  - Modified office hour schedule (to be announced)

Lecture Overview

- Object Oriented Programming (OOP)
- Abstractions (Maps project)

Object-Oriented Programming (OOP)

- Main idea
  - Encapsulate both data and behavior together
- Most modern programming languages support OOP
  - Python, Java, C++, etc.
- What does it look like?



```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def have_birthday(self):
        self.age += 1
        print("Happy birthday!", self.name, "is now:", self.age)
        return self.age
```

```
>>> paul = Person("Paul", 27)
>>> ringo = Person("Ringo", 29)
>>> ringo.age
30
```

`ringo` is a Person instance that contains both data/state (`name`, `age`) and behavior (methods like: `have_birthday()`)

(Demo: 12.py:Demo00)

Object-Oriented Programming is About *Design*

"In my version of computational thinking, I imagine an abstract machine with just the data types and operations that I want. If this machine existed, then I could write the program I want.

But it doesn't. Instead I have introduced a bunch of subproblems — the data types and operations — and I need to figure out how to implement them. I do this over and over until I'm working with a real machine or a real programming language. That's the art of design."

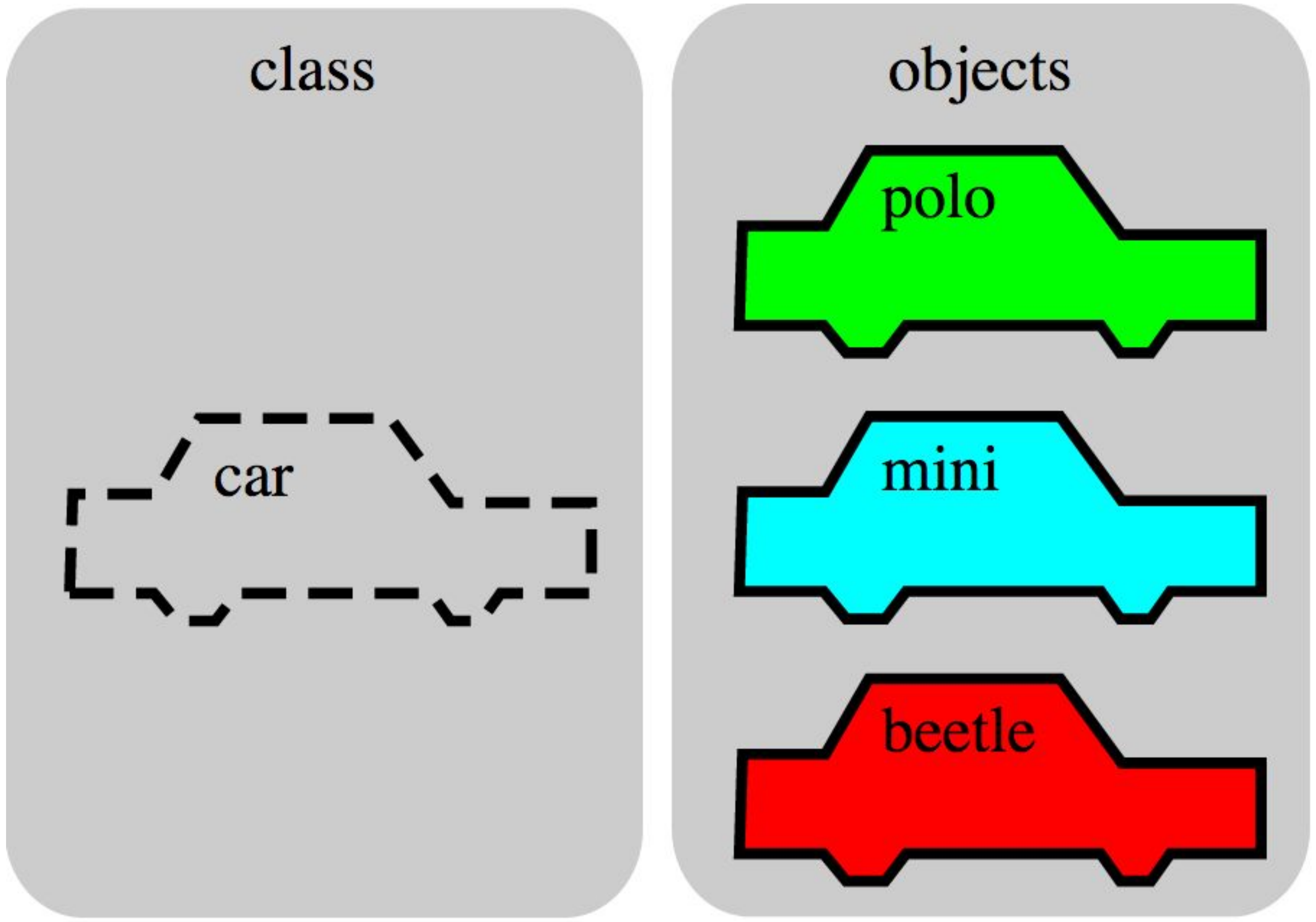
— Barbara Liskov,

Turing Award Winner, UC Berkeley '61.

Full interview

Objects

- An **object** is the instance of a class.



Analogy:
A "class" is a "blueprint", or "factory"

An "object" is an actual "built/instantiated" entity based off the blueprint (class)

In the previous slide:
\`Person\` is the **class**,
and the \`ringo\` **object**
is an **instance** of the
\`Person\` class.

Objects

- Objects are concrete instances of classes in memory.
- They have *state*
  - mutable vs immutable (lists vs tuples)
- Methods are functions that belong to an object
  - Objects do a collection of **related** things
- In Python, *everything* is an object
  - All objects have attributes
  - Manipulation happens through methods

```
# Example: `list` is a class!
>>> list
<class 'list'>
# `my_nums` is a `list` instance
>>> my_nums = [1, 2, 3]
>>> type(my_nums)
<class 'list'>
# call the `list.append()` method
>>> my_nums.append(42)
```

Class Statements

Classes

A class describes the behavior of its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

Idea: All bank accounts share a withdraw method and a deposit method

```
>>> a = Account('John')
>>> a.holder
'John'

>>> a.balance
0

>>> a.deposit(15)
15

>>> a.withdraw(10)
5

>>> a.balance
5

>>> a.withdraw(10)
'Insufficient funds'
```

balance and holder are ***attributes***

deposit and withdraw are ***methods***

The Account Class

Aka "**constructor**"

`__init__` is a special method name for the function that constructs an Account instance

`self` is the instance of the Account class on which deposit was invoked: a.deposit(10)

```
class Account:

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

```
>>> a = Account('John')
>>> a.holder
'John'
>>> a.balance
0
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

Methods are functions defined in a class statement

(Demo: 12.py:Demo01)

Discussion Question: Create Many Accounts

Write a function **create** that takes a list of strings called **names**. It returns a dictionary in which each name is a key, and its value is a new **Account** with that name as the **holder**. Deposit $5 in each account before returning.

```
def create(names):
    """Creates a dictionary of accounts, each with an initial deposit of 5.

    >>> accounts = create(['Alice', 'Bob', 'Charlie'])
    >>> accounts['Alice'].holder
    'Alice'
    >>> accounts['Bob'].balance
    5
    >>> accounts['Charlie'].deposit(10)
    15
    """
    result = {name: Account(name) for name in names}
    for a in result.values():
        a.deposit(5)
    return result
```

```
# (for reference)
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    ...
```

Another Class example: the Point class

```python
class Point:
    # Constructor
    def __init__(self, x, y):
        self.x = x  # instance vars
        self.y = y
    # Getters
    def get_x(self):
        return self.x
    def get_y(self):
        return self.y
    # Instance Methods
    def distance_l2(self, pt_other):
        # Returns the L2 distance between myself
        # (self) and `pt_other`.
       return ((self.x - pt_other.x) ** 2
               + (self.y - pt_other.y) ** 2) ** 0.5
```

```
>>> pt1 = Point(1, 2)

# access instance variables either via
# getter methods, or directly
>>> pt1.get_x()
1
>>> pt1.x
1

# call a method on `pt1`
>>> pt2 = Point(3, 4)
>>> pt1.distance_l2(pt2)
2.8284271247461903
```

Question: implement the `distance_L2()` method.

Recall: distance_L2 = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Hint: use the `sqrt()` function:

```
>>> from math import sqrt
```

Or: use `num ** 0.5` as sqrt(num).

Abstractions ("Maps" project)

- In Project01 ("Maps"), you are asked to work with the `restaurant` and `user` abstract data type, along with selector and constructor functions, and an "abstraction barrier"
-

Example: the point ADT

Suppose we wanted to define a “2d point” data type. A 2d point has an x coordinate, and a y coordinate.

Question: in Python (without using OOP), how would you represent a 2d point?

Answer: Let’s represent a 2d point as a list with two elements: [int x, int y]

Note: there are many ways you could have implemented this

Example: the point ADT

An “undisciplined” way of working with our “2d point” data type would be to work at the Python list level, writing code like this:

```
point_a = [1, 2]
point_b = [4, 5]

def distance_l2(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((p1[0] - p2[0]) ** 2 + (p2[1] - p2[1]) ** 2) ** 0.5

>>> print(f"dist btwn point_a and point_b: {distance_l2(point_a, point_b)}")
dist btwn point_a and point_b: 3.0
>>> print(f"x coord of point_a is: {point_a[0]}")
x coord of point_a is: 1
>>> print(f"y coord of point_a is: {point_a[1]}")
y coord of point_a is: 2
```

Example: the point ADT

- While it does work, the resulting code has the following issues:
- There is no abstraction in the `distance_l2()` function. It assumes that a point is a list [x, y], and does direct list indexing
- Aka "assumes the 2d point internal representation"

```python
point_a = [1, 2]
point_b = [4, 5]

def distance_l2(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((p1[0] - p2[0])**2 + (p2[1] - p2[1])**2)**0.5
```

This code only works if p1, p2 are lists of the format [x, y]. Brittle code.

Example: the point ADT

- What if we need to change the 2d point internal representation?
- Example: suppose we want to attach a “str color” to a point?

```
~~point_a = [1, 2]~~
~~point_b = [4, 5]~~
```

```
point_a = ["red", 1, 2]
point_b = ["blue", 4, 5]
```

```
def distance_l2(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((p1[0] - p2[0])**2 + (p2[1] - p2[1])**2)**0.5
```

Now, this function will break! We need to change (refactor) all code that uses our 2d point data type to adjust to the new internal representation

Example: the point ADT

- It may seem OK if it’s just one function, but in larger software projects, there may be literally millions of lines of code to change…

~~point_a = [1, 2]~~
~~point_b = [4, 5]~~

```
point_a = ["red", 1, 2]
point_b = ["blue", 4, 5]

def distance_l2(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((p1[0] - p2[0])**2 + (p2[1] - p2[1])**2)**0.5

def distance_l1(p1, p2):
    # ...
def norm_l2(p1, p2):
    # ...
def norm_l1(p1, p2):
    # ...
def sum_vals(p1, p2):
    # ...
def cosine_similarity(p1, p2):
    # ...
# ...
```

Oops, we’ve got our work cut out for us…
And, worse, this is tedious, manual, error-prone work…
…could we have planned better ahead to avoid this pain?

Example: the point ADT

- Idea: let’s implement `distance_l2()` in a more abstract, generic way. Notably, one that doesn’t assume the **internal representation** of the point data type.

```
~~point_a = [1, 2]~~     point_a = [“red", 1, 2]
~~point_b = [4, 5]~~     point_b = [“blue", 4, 5]

def distance_l2(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((p1[0] - p2[0])**2 + (p2[1] - p2[1])**2)**0.5
```

p1[0], p2[0] is asking for “get me the x coordinate”

Similarly, p1[1], p2[1] is asking for “get me the y coordinate”

Example: the point ADT

- Idea: let’s implement `distance_l2()` in a more abstract, generic way. Notably, one that doesn’t assume the **internal representation** of the point data type.

```
~~point_a = [1, 2]~~     point_a = [“red", 1, 2]
~~point_b = [4, 5]~~     point_b = [“blue", 4, 5]

def distance_l2_abstract(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((get_x(p1) - get_x(p2))**2 + (get_y(p2) - get_y(p2))**2)**0.5
```

So, let’s just ask via get_x()!

And, ask for y via get_y()

Example: the point ADT

- Finally, let's define the **constructor** and **selector** functions to fully spec out our point ADT

```
~~point_a = [1, 2]~~    point_a = [“red", 1, 2]
~~point_b = [4, 5]~~    point_b = [“blue", 4, 5]

def distance_l2_abstract(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((get_x(p1) - get_x(p2))**2 + (get_y(p2) - get_y(p2))**2)**0.5
```

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

```
# selectors
def get_x(point):
    return point[1]
def get_y(point):
    return point[2]
def get_color(point):
    return point[0]
```

Example: a Point ADT

```python
# constructor
def create_point(x, y, color):
    return [color, x, y]

# selectors
def get_x(point):
    return point[1]
def get_y(point):
    return point[2]
def get_color(point):
    return point[0]
```

This is the ADT. ("Under the hood", "below the abstraction barrier", etc).
It's allowed to know details about the **internal representation** of the data type, eg "a Point is implemented as a list of three elements"

```python
# Operators
def distance_l2_abstract(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((get_x(p1) - get_x(p2))**2 + (get_y(p2) - get_y(p2))**2)**0.5
```

Example: a Point ADT

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

# selectors
def get_x(point):
    return point[1]
def get_y(point):
    return point[2]
def get_color(point):
    return point[0]
```

These are the **operations** that are built on top of the abstractions defined by the ADT.
They are NOT allowed to know details about the internal representation.
Instead, they should only use the ADT's "public-facing API/spec", aka the **constructors and selectors**

This is the "abstraction barrier".
Don't cross the boundary!

```
# Operators
def distance_l2_abstract(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((get_x(p1) - get_x(p2))**2 + (get_y(p2) - get_y(p2))**2)**0.5
```

Example: a Point ADT

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

# selectors
def get_x(point):
    return point[1]
def get_y(point):
    return point[2]
def get_color(point):
    return point[0]
```

Question: we want to change the Point internal representation to be a dict, like: {"x": 1, "y": 2, "color": "red"}. Make the changes to the ADT.

```
Answer:
# constructor
def create_point(x, y, color):
    return {"x": x, "y": y, "color": color}

# selectors
def get_x(point):
    return point["x"]
def get_y(point):
    return point["y"]
def get_color(point):
    return point["color"]
```

Note that no changes are necessary to the existing operators after this dict refactor. `distance_l2_abstract()` still works!

This is the "abstraction barrier". Don't cross the boundary!

```
# Operators
def distance_l2_abstract(p1, p2):
    # Calculates L2 distance between 2d points p1, p2
    return ((get_x(p1) - get_x(p2))**2 + (get_y(p2) - get_y(p2))**2)**0.5
```