Welcome to Data C88C!

Lecture 13: Attributes
Wednesday, July 16th, 2025
Week 3
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- Midterm: congratulations on being done!
  - **Important**: if you haven't taken the midterm and you intended to, please email cs88@berkeley.edu ASAP
  - Grading will be done within ~1-2 weeks
- HW07, Lab07 due: Fri July 19th
- No Labs this week!
- Maps due: July 24th
  - (k-means is neat!)
-

Lecture Overview

- Attributes
  - Class attributes
  - Instance vs Class attributes

Method Calls

Dot Expressions

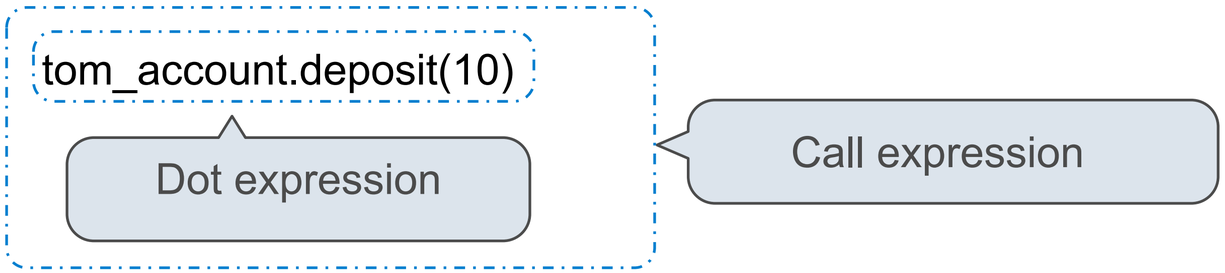
Methods are invoked using dot notation

<expression> . <name>

The <expression> can be any valid Python expression

The <name> must be a simple name

Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>



(Demo)

Attribute Lookup

Looking Up Attributes by Name

Both instances and classes have attributes that can be looked up by dot expressions

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, <name> is looked up in the class, which yields a class attribute value
4. That value is returned unless it is a function, in which case a bound method is returned instead

Discussion Question: Where's Waldo?

Question: For each class, write an expression **with no quotes or +** that evaluates to 'Waldo'

```
class Town:
    def __init__(self, w, aldo):
        if aldo == 7:
            self.street = {self.f(w): 'Waldo'}

    def f(self, x):
        return x + 1
```

Answer:

```
>>> Town(1, 7).street[2]
'Waldo'
```

```
class Beach:
    def __init__(self):
        sand = ['Wal', 'do']
        self.dig = sand.pop

    def walk(self, x):
        self.wave = lambda y: self.dig(x) + self.dig(y)
        return self
```

Reminder: s.pop(k) removes and returns the item at index k

```
>>> Beach().walk(0).wave(0)
'Waldo'
```

Class Attributes

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance

```
class Account:

    interest = 0.02    # A class attribute

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

    # Additional methods would be defined here
```

```
>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
```

The **interest** attribute is ***not*** part of the instance; it's part of the class!

(Demo)

Attribute Assignment Statements

Account class attributes

```
interest: 0.02  0.04  0.05
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

```
balance:   0
holder:    'Jim'
interest:  0.08
```

Instance attributes of tom_account

```
balance:   0
holder:    'Tom'
```

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

Discussion Question: Class Attribute Assignment

Implement the **Place** class, which takes a **name**. Its **print_history**() method prints the **name** of the **Place** and then the names of all the **Place** instances that were created before it.

```
class Place:
    last = None
    def __init__(self, n):
        self.name = n
        self.then = Place.last
        Place.last = self

    def print_history(self):
        print(self.name)
        if self.then is not None:
            self.then.print_history()
```

OK to write **self.last** or **type(self.last)**

Not ok to write **self.last**

```
>>> places = [Place(x*2) for x in range(10)]
>>> places[4].print_history()
8
6
4
2
0
>>> places[6].print_history()
12
10
8
6
4
2
0
```

Why OOP?

- OOP allows programmers to reason at a higher level of abstraction
- Ex: work with "Point" objects, rather than "a two-element list [float x, float y]"

Next lecture, we will learn an OOP technique that is not so easily handled via this "function OOP" approach: inheritance

Question: try implementing this `Point` class, but without OOP.
Hint: try defining constructor/methods as functions. What obstacles do you run into?

```
class Point:
    num_points = 0
    # Constructor
    def __init__(self, x, y):
        self.x, self.y = x, y  # instance vars
        Point.num_points += 1  # class var
    # 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
```

(Demo: 13.py:Demo00)

Answer: here's one way:

```
# global var
POINT_CLASS_VARS = {"num_points": 0}
def point_constructor(x, y):
    POINT_CLASS_VARS["num_points"] += 1
    return {"x": x, "y": y}
def point_get_x(pt):
    return pt["x"]
def point_get_y(pt):
    return pt["y"]
def point_get_num_points():
    return POINT_CLASS_VARS["num_points"]
def point_distance_l2(pt_a, pt_b):
    return ((point_get_x(pt_a) - point_get_x("x"))
** 2 + (point_get_y(pt_a) - point_get_y(pt_b)) **
2) ** 0.5
```