Objects and Attributes

Announcements

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
```

balance and holder are ***attributes***

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

deposit and withdraw are ***methods***

The Account Class

```
class Account:
```

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

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

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

```
    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
```

Methods are functions defined in a class statement

(Demo)

```
>>> 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'
```

Practice 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
```

Method Calls

Dot Expressions

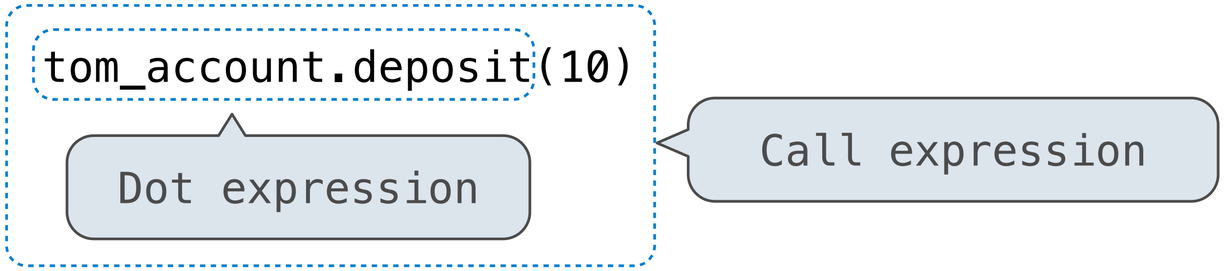
Methods are invoked using dot notation

<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)

Functions vs Bound Methods

(Demo)

Classes as Values

(Demo)

Break: 5 minutes

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

(Demo)

Practice Question: Where's Waldo?

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
```

```
>>> 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!

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
```

Practice 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
```