

Inheritance

Announcements

Attributes

Methods and Functions

Python distinguishes between:

- *Functions*, which we have been creating since the beginning of the course, and
- *Bound methods*, which couple together a function and the object on which that method will be invoked

Object + Function = Bound Method

```
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>
```

```
>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1004)
2015
```

Function: all arguments within parentheses

Method: One object before the dot and other arguments within parentheses

Terminology: Attributes, Functions, and Methods

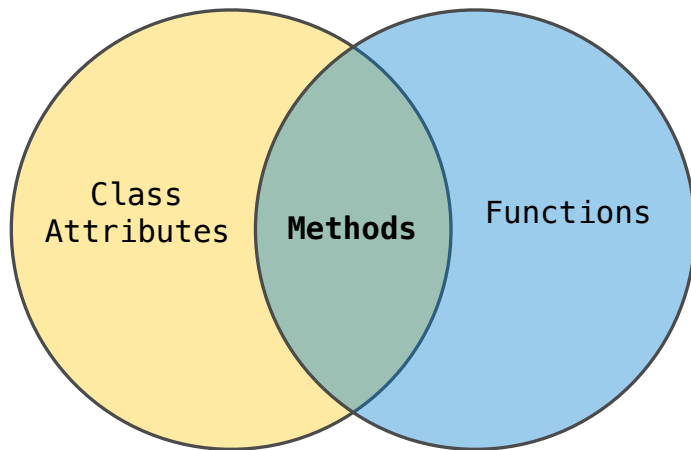
All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance

Class attribute: attribute of the class of an instance

Terminology:



Python object system:

Functions are objects

Bound methods are also objects: a function that has its first parameter "self" already bound to an instance

Dot expressions evaluate to bound methods for class attributes that are functions

`<instance>.<method_name>`

Looking Up Attributes by Name

`<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

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

Assignment to Attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```

Instance
Attribute
Assignment :

tom_account.interest = 0.08

This expression
evaluates to an
object

But the name ("interest")
is not looked up

Attribute
assignment
statement adds
or modifies the
attribute named
"interest" of
tom_account

Class
Attribute
Assignment :

Account.interest = 0.04

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
```

attribute isn't locked
after assigned

```
>>> 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
```

Inheritance

Inheritance

Inheritance is a technique for relating classes together

A common use: Two similar classes differ in their degree of specialization

The specialized class may have the same attributes as the general class, along with some special-case behavior

```
class <Name>(<Base Class>):  
    <suite>
```

Conceptually, the new subclass inherits attributes of its base class

The subclass may override certain inherited attributes

Using inheritance, we implement a subclass by specifying its differences from the the base class

Inheritance Example

A `CheckingAccount` is a specialized type of `Account`

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest      # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)   # Deposits are the same
20
>>> ch.withdraw(5)   # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class `Account`

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
        or
        return super().withdraw(amount + self.withdraw_fee)
```

class method needs a self argument

Looking Up Attribute Names on Classes

Base class attributes *aren't* copied into subclasses!

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom') # Calls Account.__init__
>>> ch.interest                 # Found in CheckingAccount
0.01
>>> ch.deposit(20)              # Found in Account
20
>>> ch.withdraw(5)              # Found in CheckingAccount
14
```

(Demo)

Object-Oriented Design

Designing for Inheritance

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects

Look up attributes on instances whenever possible

```
class CheckingAccount(Account):  
    """A bank account that charges for withdrawals."""  
    withdraw_fee = 1  
    interest = 0.01  
    def withdraw(self, amount):  
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Attribute look-up
on base class

Preferred to `CheckingAccount.withdraw_fee`
to allow for specialized accounts

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

- E.g., a checking account is a specific type of account
- So, CheckingAccount inherits from Account

Composition is best for representing has-a relationships

- E.g., a bank has a collection of bank accounts it manages
- So, A bank has a list of accounts as an attribute

(Demo)

```
43
44 class Bank:
45     """A bank *has* accounts.
46
47     >>> bank = Bank()
48     >>> john = bank.open_account('John', 10)
49     >>> jack = bank.open_account('Jack', 5, CheckingAccount)
50     >>> john.interest
51     0.02
52     >>> jack.interest
53     0.01
54     >>> bank.pay_interest()
55     >>> john.balance
56     10.2
57     >>> bank.too_big_to_fail()
58     True
59     """
60     def __init__(self):
61         self.accounts = []
62
63     def open_account(self, holder, amount, kind=Account):
64         account = kind(holder)
65         account.deposit(amount)
66         self.accounts.append(account)
67         return account
68
69     def pay_interest(self):
70         for a in self.accounts:
71             a.deposit(a.balance * a.interest)
72
73     def too_big_to_fail(self):
74         return len(self.accounts) > 1
```



Attributes Lookup Practice

Inheritance and Attribute Lookup

```
class A:
    z = -1
    def f(self, x):
        return B(x-1)
```

```
class B(A):
    n = 4
    def __init__(self, y):
        if y:
            self.z = self.f(y)
        else:
            self.z = C(y+1)
```

```
class C(B):
    def f(self, x):
        return x
```

```
a = A()
b = B(1)
b.n = 5
```

```
>>> C(2).n
```

4

```
>>> a.z == C.z
```

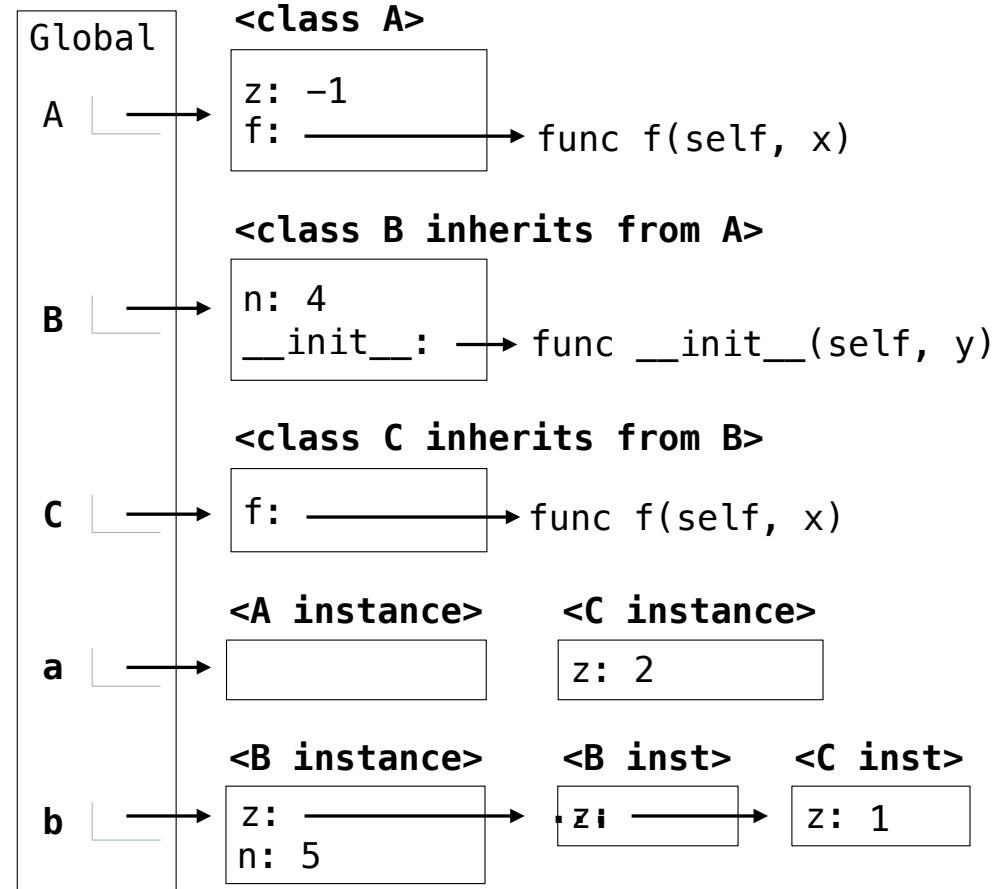
True

```
>>> a.z == b.z
```

False

Which evaluates
to an integer?

b.z
b.z.z
b.z.z.z
b.z.z.z.z
None of these



Multiple Inheritance

Multiple Inheritance

```
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)
```

A class may inherit from multiple base classes in Python

CleverBank marketing executive has an idea:

- Low interest rate of 1%
- A \$1 fee for withdrawals
- A \$2 fee for deposits
- A free dollar when you open your account

```
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1 # A free dollar!
```

Multiple Inheritance

A class may inherit from multiple base classes in Python.

```
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):  
    def __init__(self, account_holder):  
        self.holder = account_holder  
        self.balance = 1                # A free dollar!
```

Instance attribute

```
>>> such_a_deal = AsSeenOnTVAccount('John')
```

```
>>> such_a_deal.balance
```

```
1
```

SavingsAccount method

```
>>> such_a_deal.deposit(20)
```

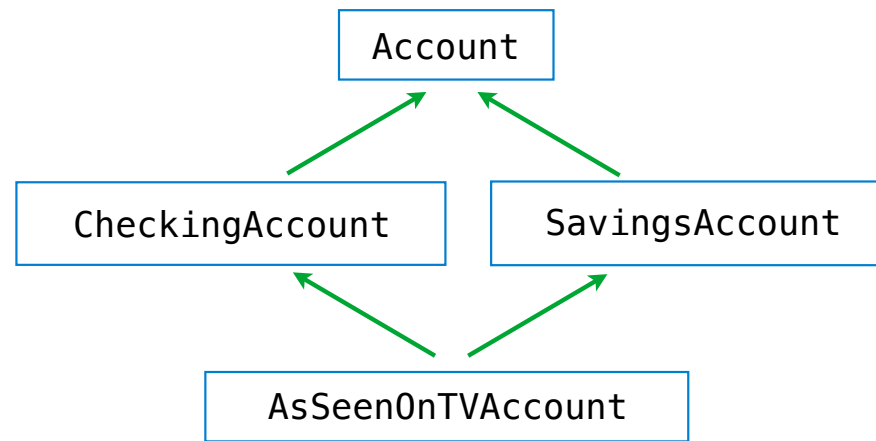
```
19
```

CheckingAccount method

```
>>> such_a_deal.withdraw(5)
```

```
13
```

Resolving Ambiguous Class Attribute Names



Instance attribute

```
>>> such_a_deal = AsSeenOnTVAccount('John')
```

```
>>> such_a_deal.balance
```

```
1
```

SavingsAccount method

```
>>> such_a_deal.deposit(20)
```

```
19
```

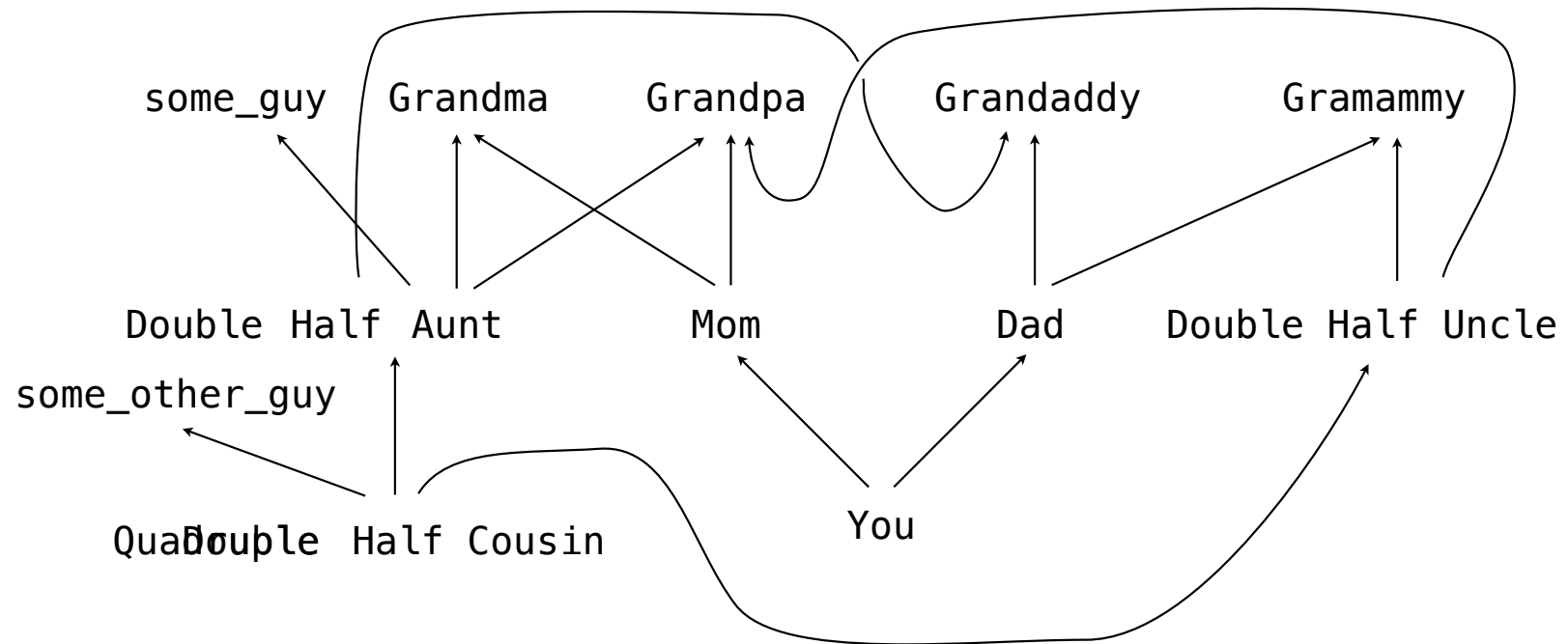
CheckingAccount method

```
>>> such_a_deal.withdraw(5)
```

```
13
```


Complicated Inheritance

Biological Inheritance



Moral of the story: Inheritance can be complicated, so don't overuse it!