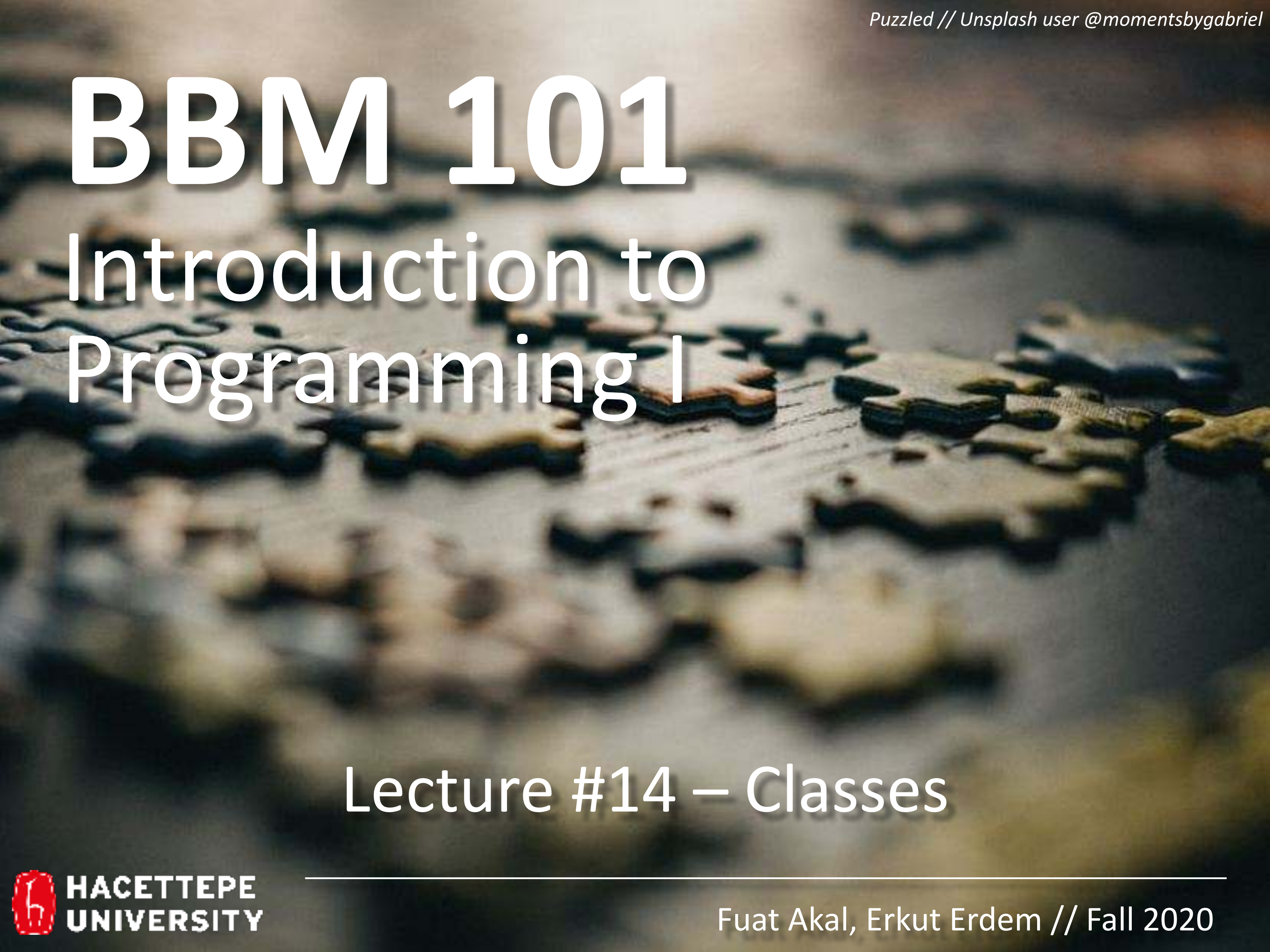


Puzzled // Unsplash user @momentsbygabriel

BBM 101

Introduction to Programming I

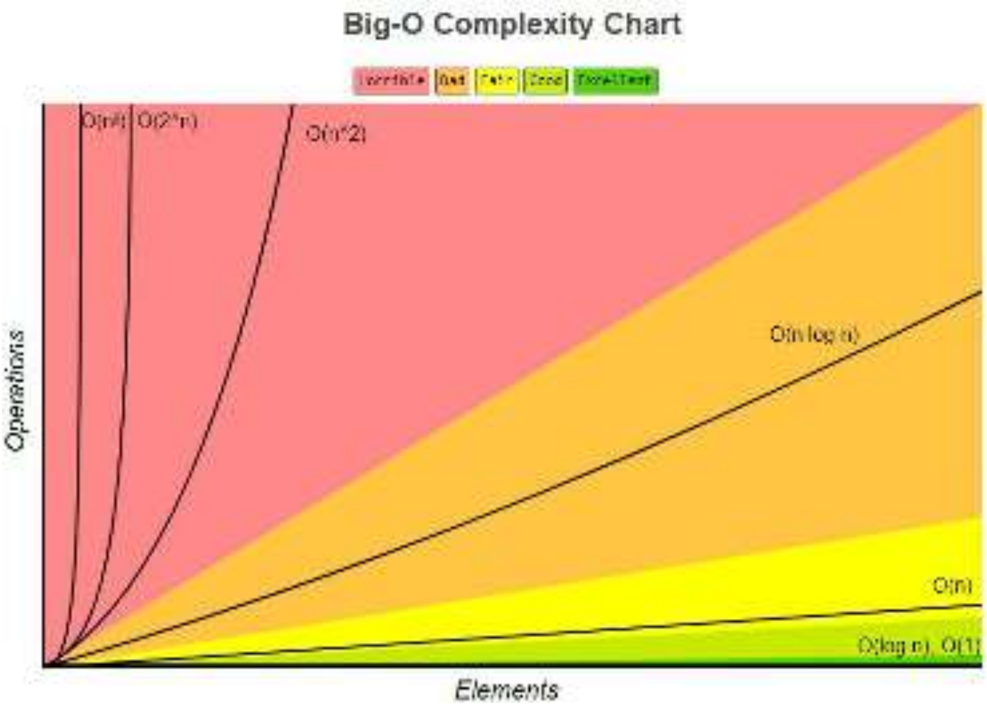
Lecture #14 – Classes

HACETTEPE UNIVERSITY

Fuat Akal, Erkut Erdem // Fall 2020

Last time… **Algorithmic Speed**

- ***O(1)*** denotes constant running time
- ***O(log n)*** denotes logarithmic running time
- ***O(n)*** denotes linear running time
- ***O(n log n)*** denotes log-linear running time
- ***O(n^c)*** denotes polynomial running time
  (*c* is a constant)
- ***O(c^n)*** denotes exponential running time
  (*c* is a constant being raised to a power based on size of input)



Lecture Overview

- Classes
  - Object Oriented Programming
  - Class Statements
  - Methods
  - Attributes
  - Inheritence

Disclaimer: Much of the material and slides for this lecture were borrowed from John DeNero’s Berkeley CS 61A

Lecture Overview

- Classes
  - Object Oriented Programming
  - Class Statements
  - Methods
  - Attributes
  - Inheritence

Object-Oriented Programming

- A method for organizing programs
  - Data abstraction
  - Bundling together information and related behavior

- A metaphor for computation using distributed state Each object has its own local state
  - Each object also knows how to manage its own local state, based on method calls
  - Method calls are messages passed between objects
  - Several objects may all be instances of a common type
  - Different types may relate to each other

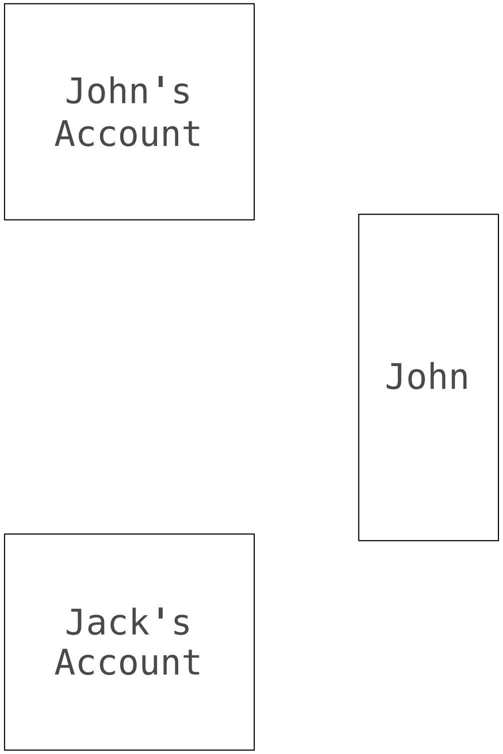
- Specialized syntax & vocabulary to support this metaphor

Object-Oriented Programming

- A method for organizing programs
  - Data abstraction
  - Bundling together information and related behavior

- A metaphor for computation using distributed state Each object has its own local state
  - Each object also knows how to manage its own local state, based on method calls
  - Method calls are messages passed between objects
  - Several objects may all be instances of a common type
  - Different types may relate to each other

- Specialized syntax & vocabulary to support this metaphor

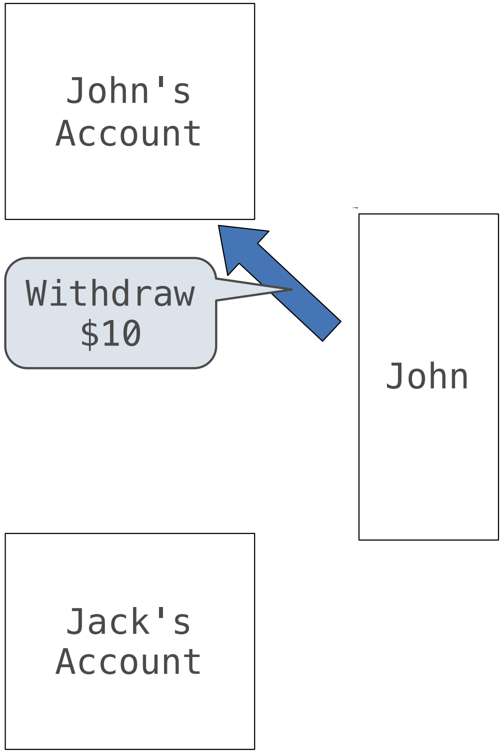


Object-Oriented Programming

- A method for organizing programs
  - Data abstraction
  - Bundling together information and related behavior

- A metaphor for computation using distributed state Each object has its own local state
  - Each object also knows how to manage its own local state, based on method calls
  - Method calls are messages passed between objects
  - Several objects may all be instances of a common type
  - Different types may relate to each other

- Specialized syntax & vocabulary to support this metaphor



Object-Oriented Programming

- A method for organizing programs

– Data abstraction

– Bundling together information and related behavior

- A metaphor for computation using distributed state Each object has its own local state

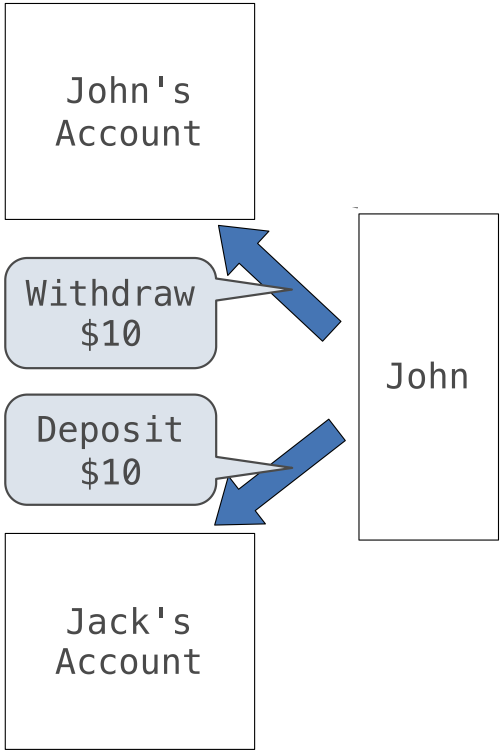
– Each object also knows how to manage its own local state, based on method calls

– Method calls are messages passed between objects

– Several objects may all be instances of a common type

– Different types may relate to each other

- Specialized syntax & vocabulary to support this metaphor



Object-Oriented Programming

- A method for organizing programs

– Data abstraction

– Bundling together information and related behavior

- A metaphor for computation using distributed state Each object has its own local state

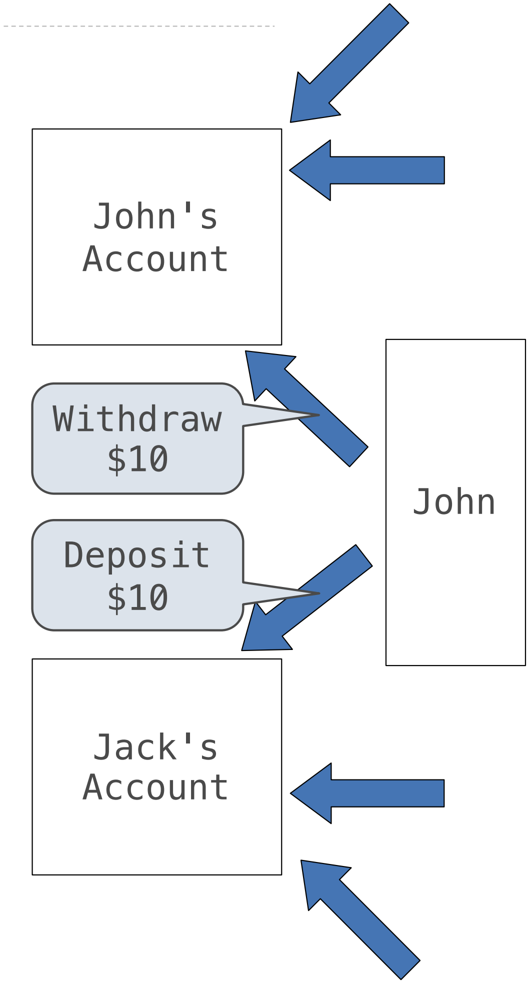
– Each object also knows how to manage its own local state, based on method calls

– Method calls are messages passed between objects

– Several objects may all be instances of a common type

– Different types may relate to each other

- Specialized syntax & vocabulary to support this metaphor



Object-Oriented Programming

- A method for organizing programs

– Data abstraction

– Bundling together information and related behavior

- A metaphor for computation using distributed state Each object has its own local state

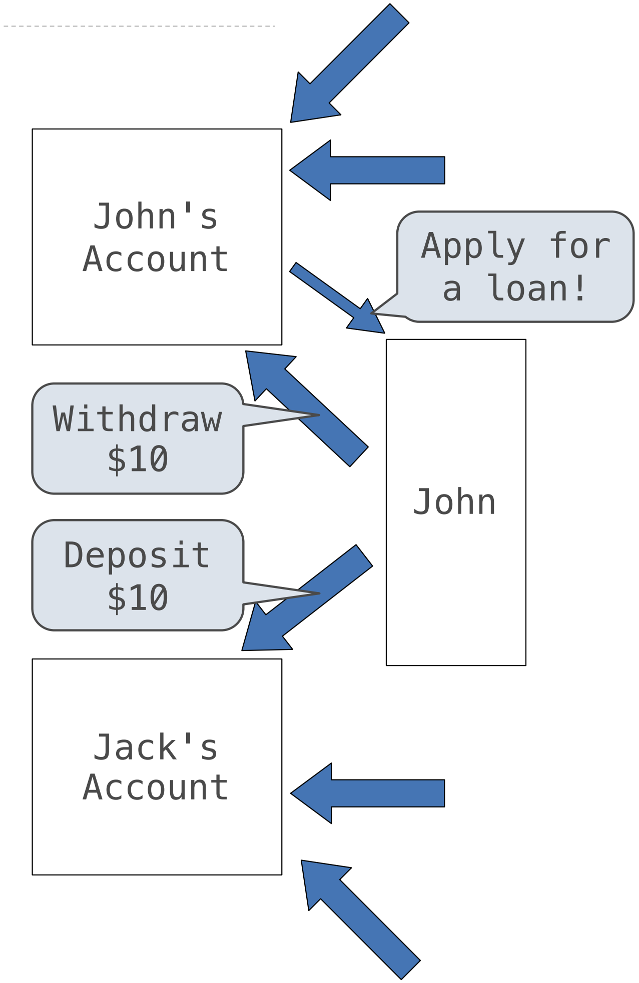
– Each object also knows how to manage its own local state, based on method calls

– Method calls are messages passed between objects

– Several objects may all be instances of a common type

– Different types may relate to each other

- Specialized syntax & vocabulary to support this metaphor



Classes

- A class serves as a template for 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

```
>>> a = Account('John’)
>>> a.holder
'John'
>>> a.balance
0
```

Idea: All bank accounts should have `withdraw` and `deposit` behaviors that all work in the same way

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

Better idea: All bank accounts share a `withdraw` method and a `deposit` method

Lecture Overview

- Classes
  - Object Oriented Programming
  - Class Statements
  - Methods
  - Attributes
  - Inheritence

The Class Statement

```
class <name>:
    <suite>
```

The `suite` is executed when
the class statement is executed.

- A `class` statement creates a new class and binds that class to `<name>` in the first frame of the current environment
- Assignment & `def` statements in `<suite>` create attributes of the class (not names in frames)

```
>>> class Clown:
...     nose = 'big and red'
...     def dance():
...         return 'No thanks’
...
>>> Clown.nose
'big and red’
>>> Clown.dance()
'No thanks'
>>> Clown
<class '__main__.Clown'>
```

Object Construction

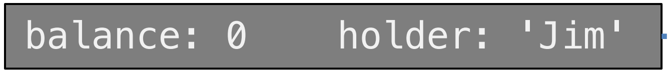
- **Idea**: All bank accounts have a `balance` and an account `holder`; the `Account` class should add those attributes to each of its instances

```
>>> a = Account('Jim’)
>>> a.holder
'Jim'
>>> a.balance
0
```

- When a class is called:

1. A new instance of that class is created:
2. The `__init__` method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression

An account instance



balance: 0   holder: 'Jim'

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

__init__ is called a constructor

Object identity

- Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John’)
>>> b = Account('Jack’)
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to `Account` creates a new `Account` instance. There is only one `Account` class.

- Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```
>>> a is a
True
>>> a is not b
True
```

- Binding an object to a new name using assignment does not create a new object:

```
>>> c = a
>>> c is a
True
```

Lecture Overview

- Classes
  - Object Oriented Programming
  - Class Statements
  - Methods
  - Attributes
  - Inheritence

Methods

- Methods are functions defined in the suite of a class statement

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

self should always be bound to an instance of the Account class

```
    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
```

- These def statements create function objects as always, but their names are bound as attributes of the class

Invoking Methods

- All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

```
class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Defined with two parameters

- Dot notation automatically supplies the first argument to a method

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Bound to self

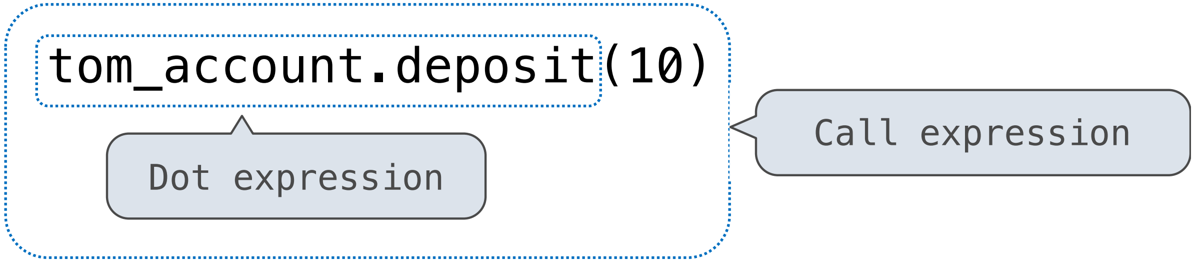
Invoked with one argument

Dot expressions

- Objects receive messages via dot notation
- Dot notation accesses attributes of the instance or its class

```
<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>



```
tom_account.deposit(10)
```

Lecture Overview

- Classes
  - Object Oriented Programming
  - Class Statements
  - Methods
  - Attributes
  - Inheritence

Accessing Attributes

- Using getattr, we can look up an attribute using a string

```
>>> getattr(tom_account, 'balance’)
10
>>> hasattr(tom_account, 'deposit’)
True
```

- getattr and dot expressions look up a name in the same way
- Looking up an attribute name in an object may return:
  - One of its instance attributes, or
  - One of the attributes of its class

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
```

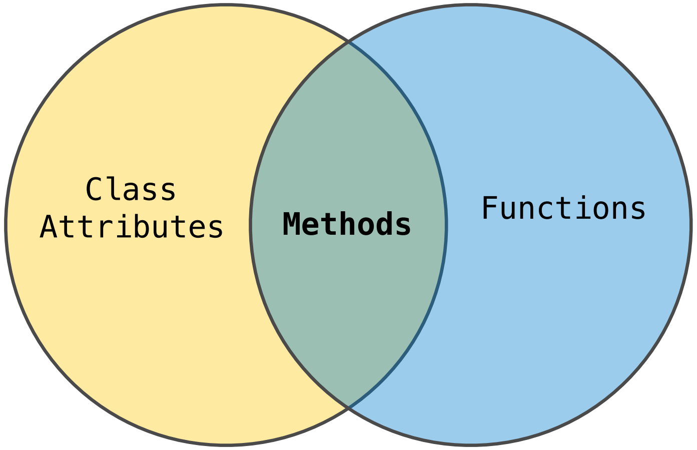
Function: all arguments within parentheses

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!

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

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

tom_account = Account('Tom')
```

Class Attribute Assignment:

```
Account.interest = 0.04
```

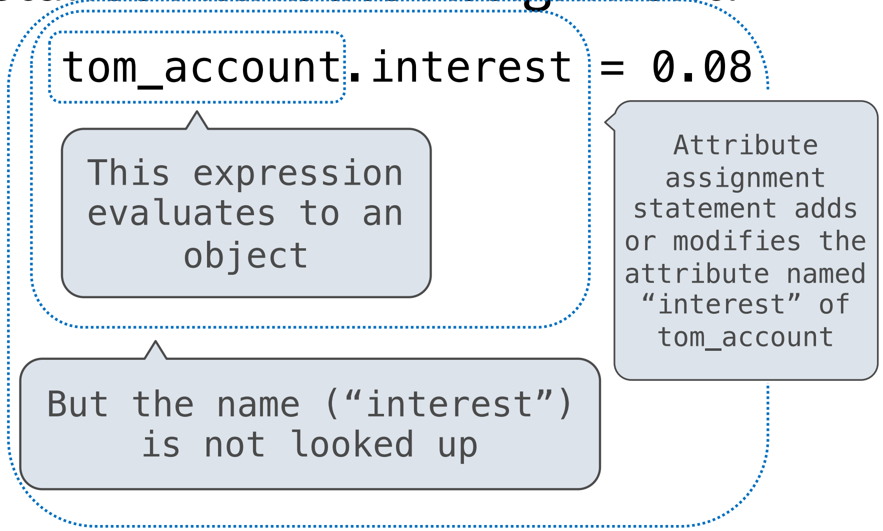
Assignment to Attributes

- Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression
  - If the object is a class, then assignment sets a class attribute
  - If the object is an instance, then assignment sets an instance attribute

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

tom_account = Account('Tom')
```

Instance Attribute Assignment:



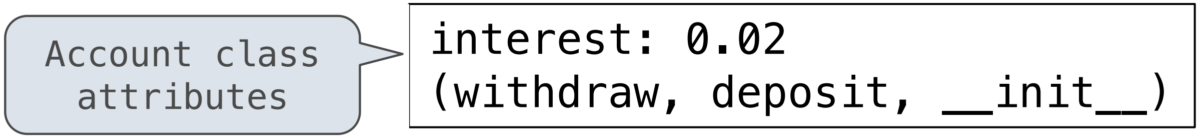
```
tom_account.interest = 0.08
```

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

Attribute Assignment Statements

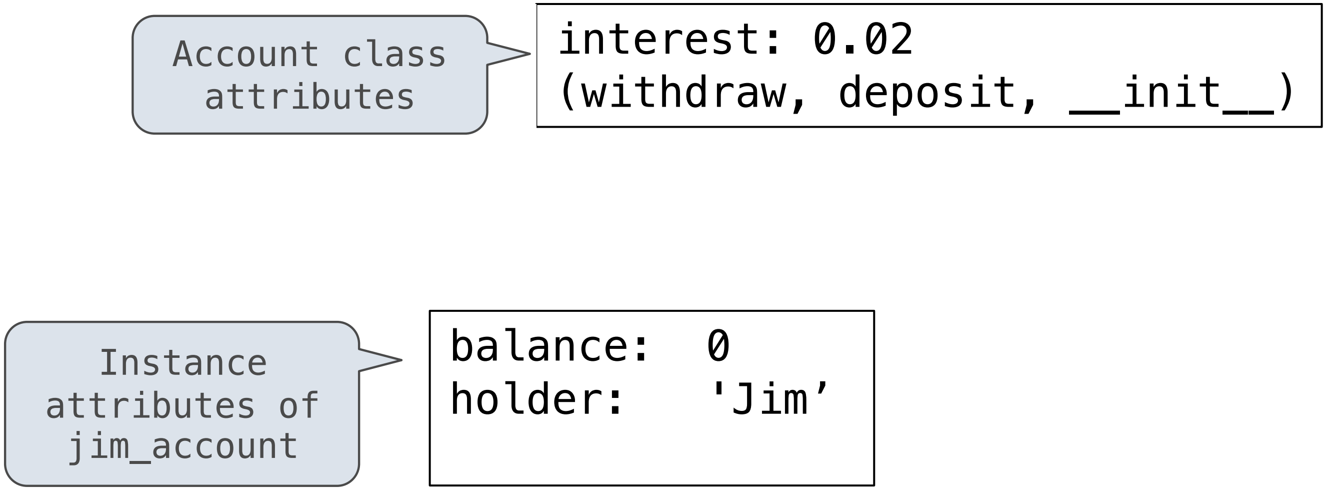


Account class attributes

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

```
>>> jim_account = Account('Jim')
```

Attribute Assignment Statements



Account class attributes

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

Instance attributes of jim_account

```
balance:  0
holder:   'Jim’
```

```
>>> jim_account = Account('Jim')
```

Attribute Assignment Statements

Account class attributes

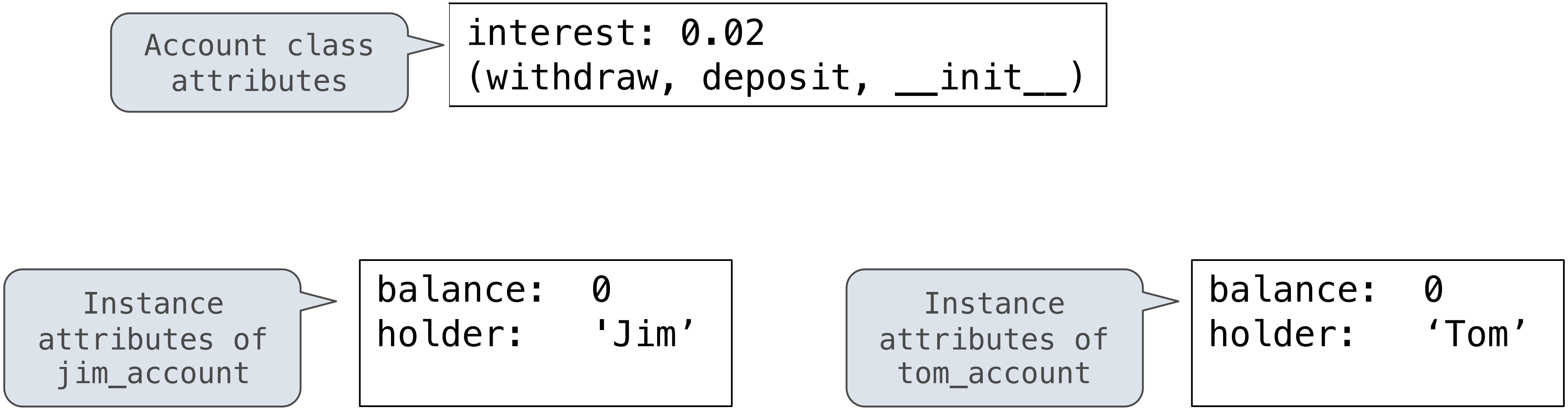
```
interest: 0.02
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

```
balance:   0
holder:    'Jim’
```

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
```

Attribute Assignment Statements



Account class attributes

interest: 0.02
(withdraw, deposit, __init__)

Instance attributes of jim_account

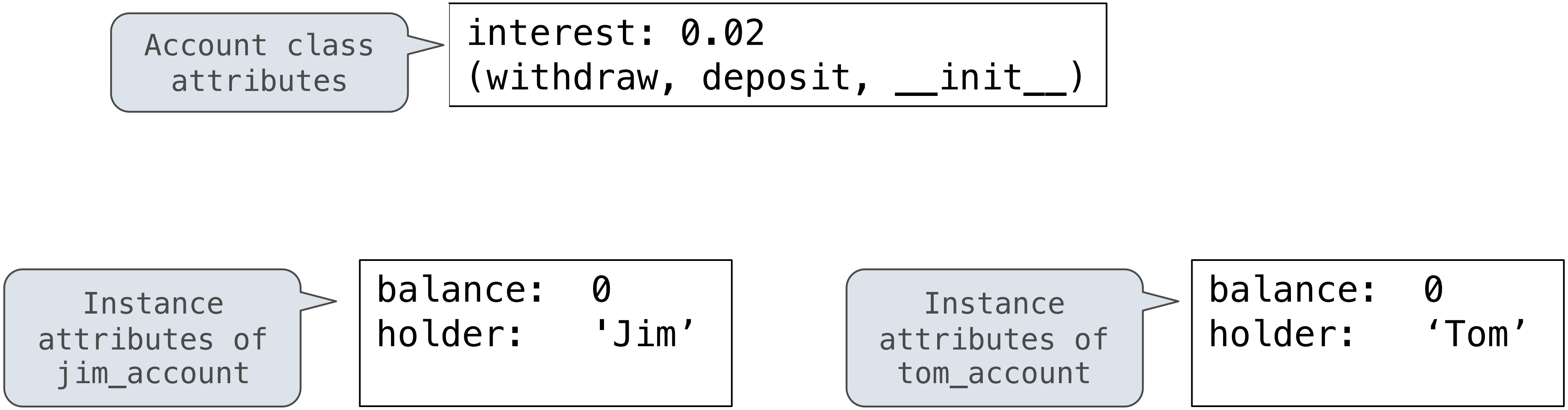
balance: 0
holder: 'Jim’

Instance attributes of tom_account

balance: 0
holder: ‘Tom’

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
```

Attribute Assignment Statements



Account class attributes

interest: 0.02
(withdraw, deposit, __init__)

Instance attributes of jim_account

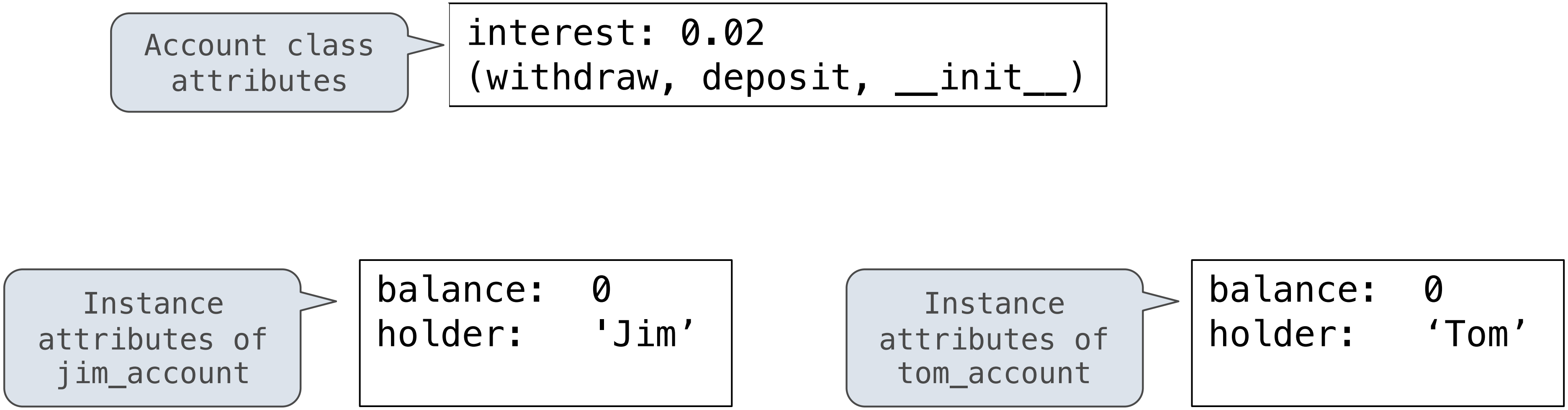
balance: 0
holder: 'Jim’

Instance attributes of tom_account

balance: 0
holder: ‘Tom’

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

Attribute Assignment Statements



Account class attributes

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

Instance attributes of jim_account

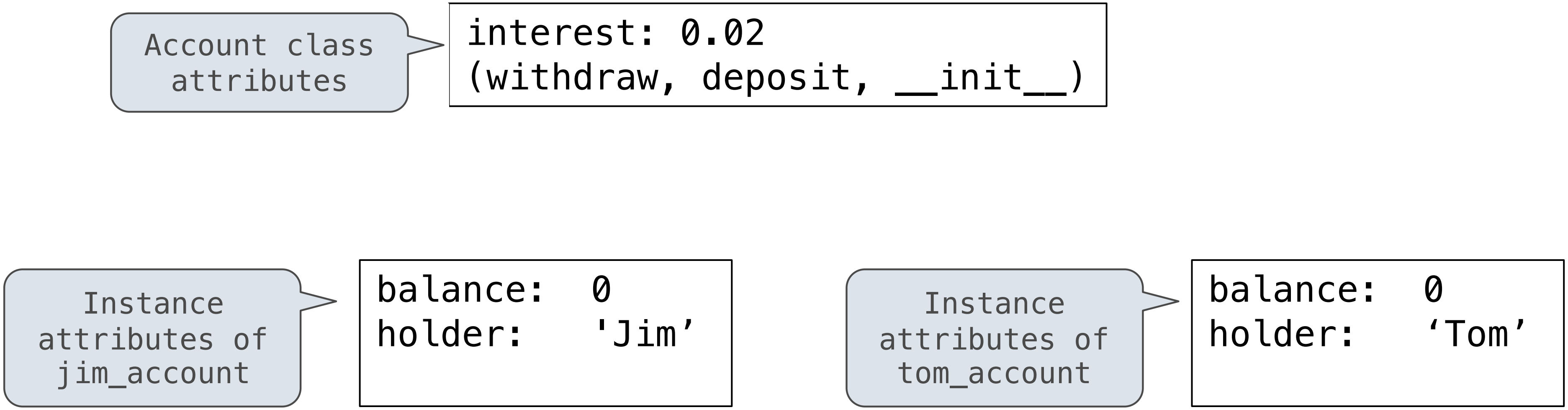
```
balance:   0
holder:    'Jim’
```

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
```

Attribute Assignment Statements



Account class attributes

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

Instance attributes of jim_account

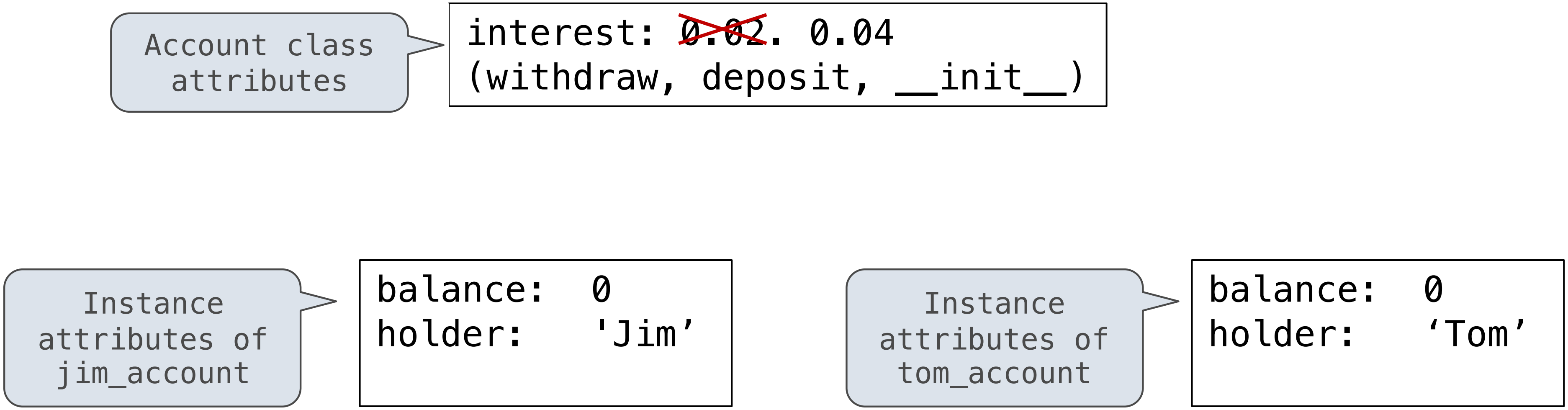
```
balance:   0
holder:    'Jim’
```

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
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~. 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

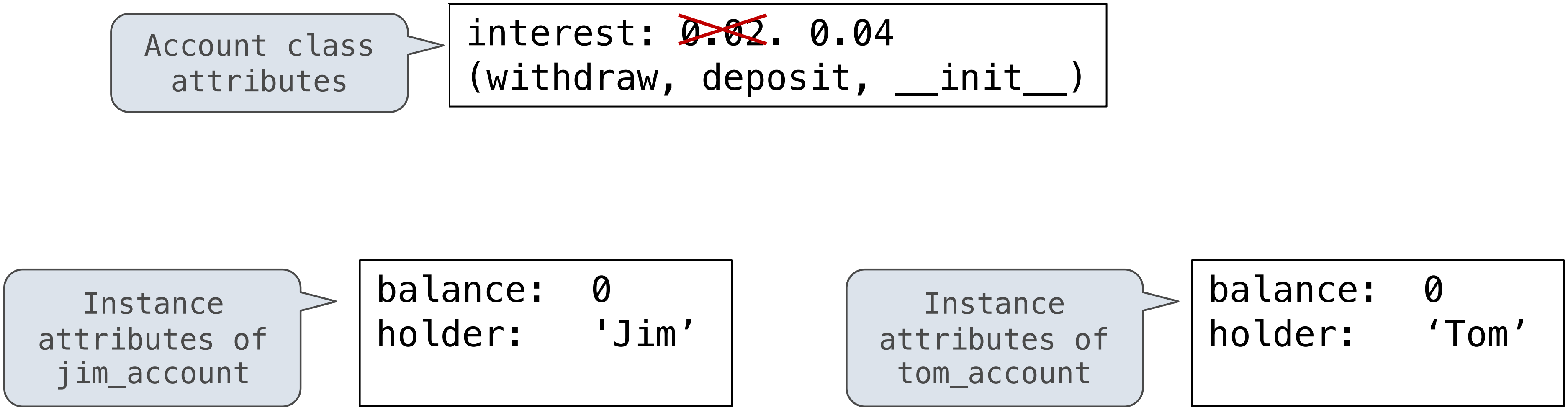
balance: 0
holder: 'Jim’

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
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~. 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

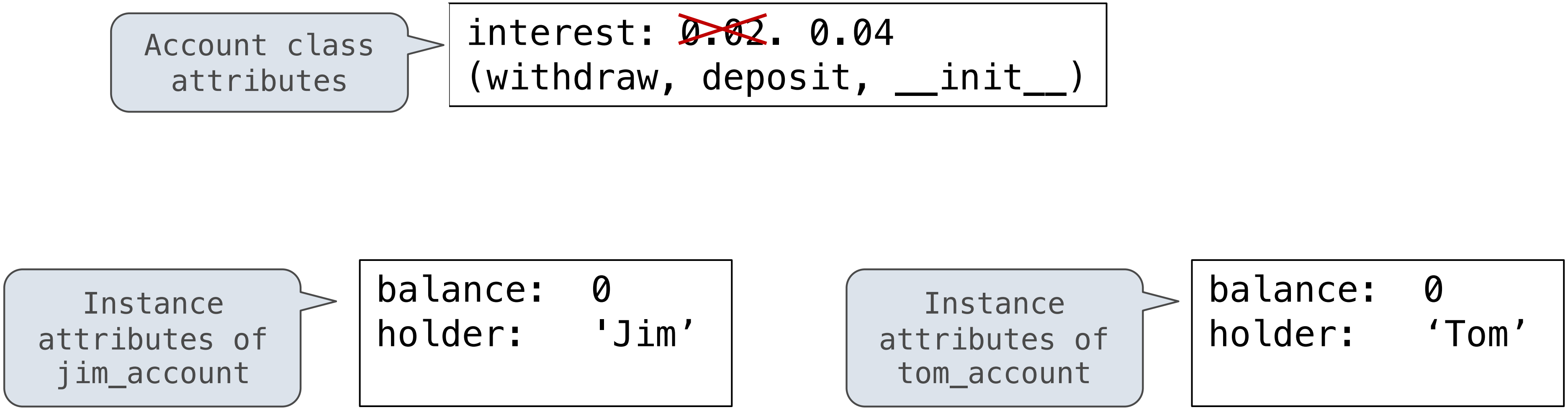
balance: 0
holder: 'Jim’

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
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~. 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

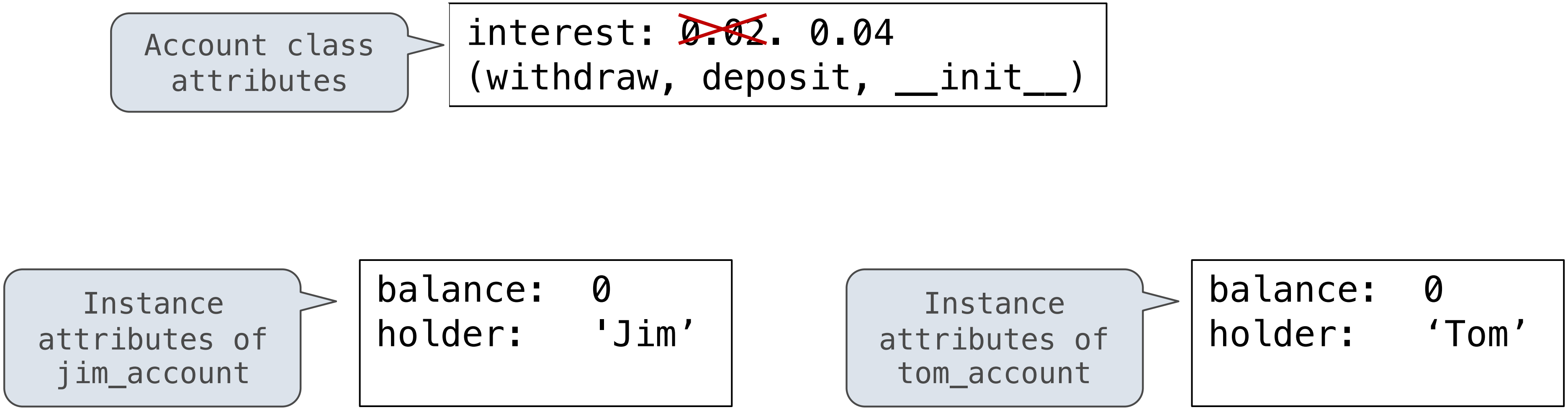
balance: 0
holder: 'Jim’

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 Assignment Statements



Account class attributes

interest: ~~0.02~~. 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim’

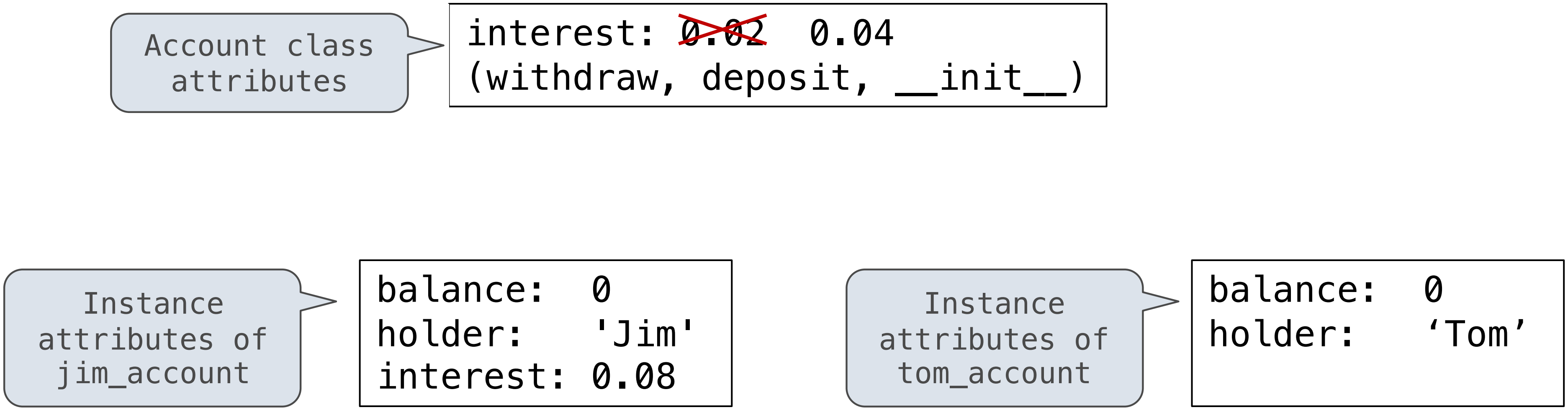
Instance attributes of tom_account

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
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

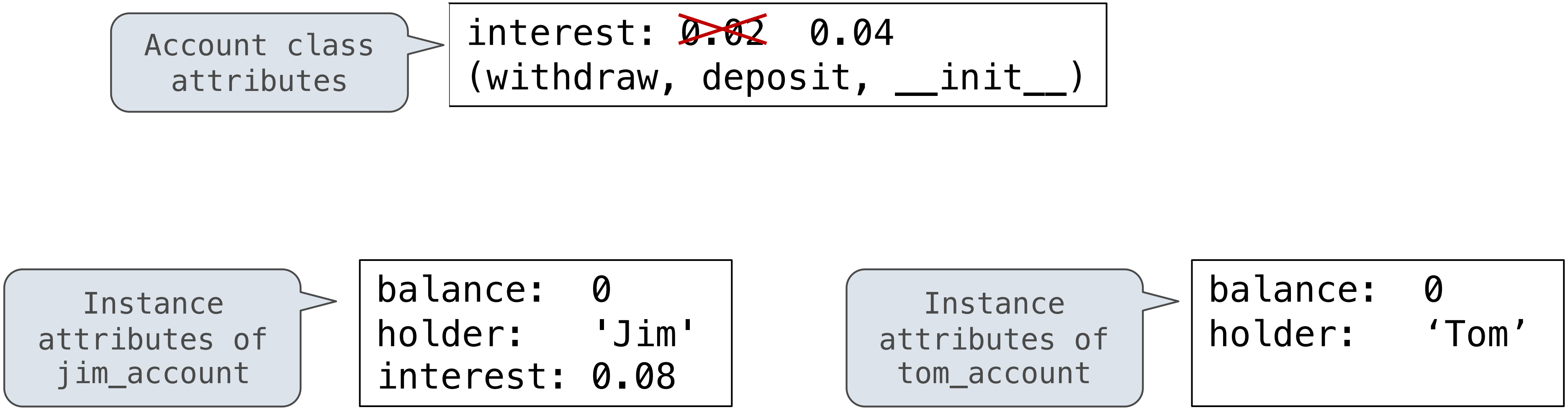
Instance attributes of tom_account

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
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

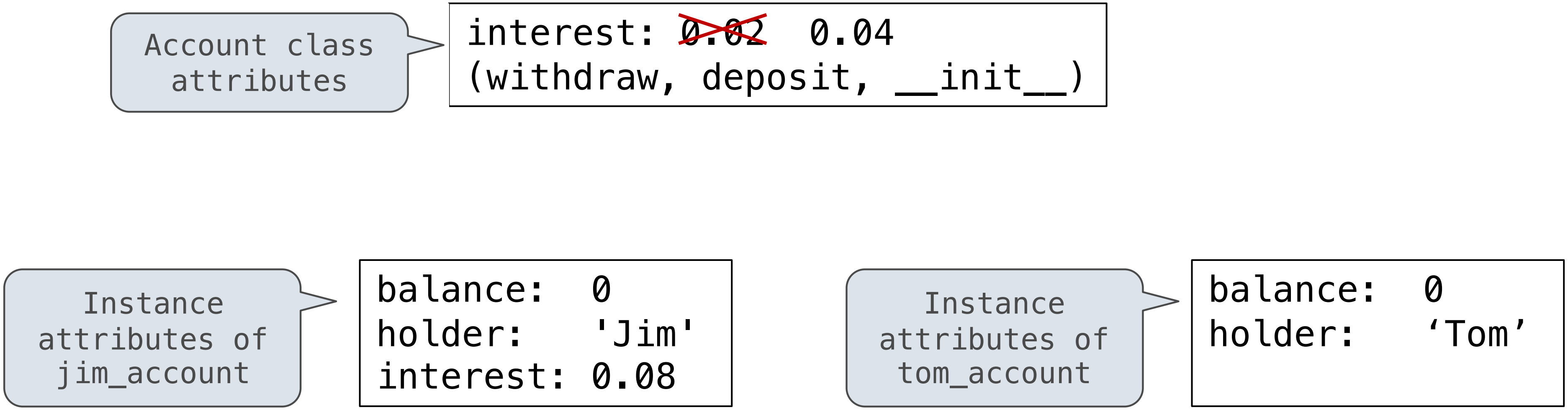
Instance attributes of tom_account

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
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

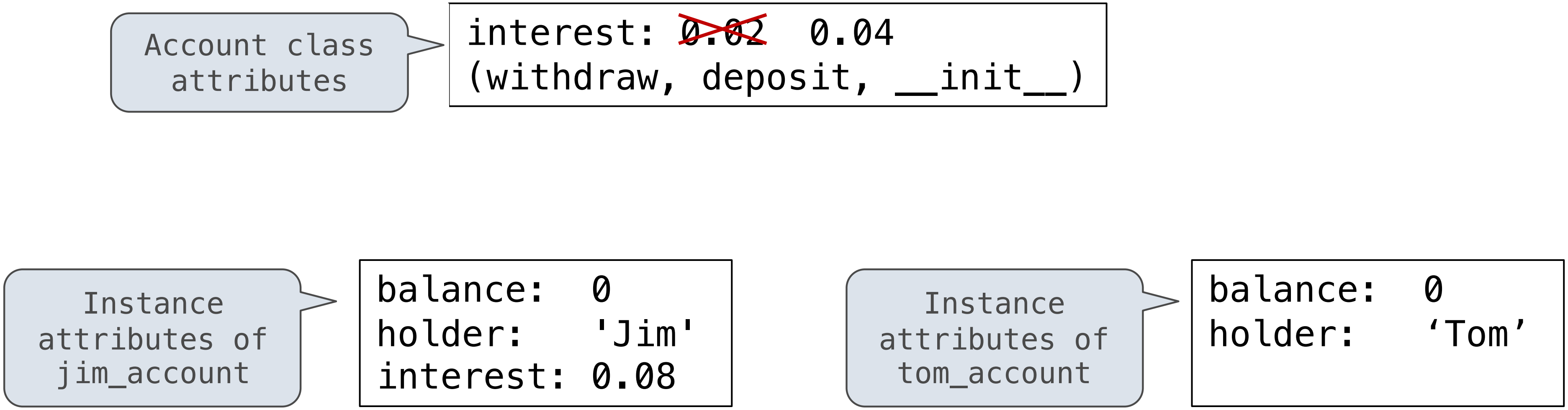
Instance attributes of tom_account

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
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

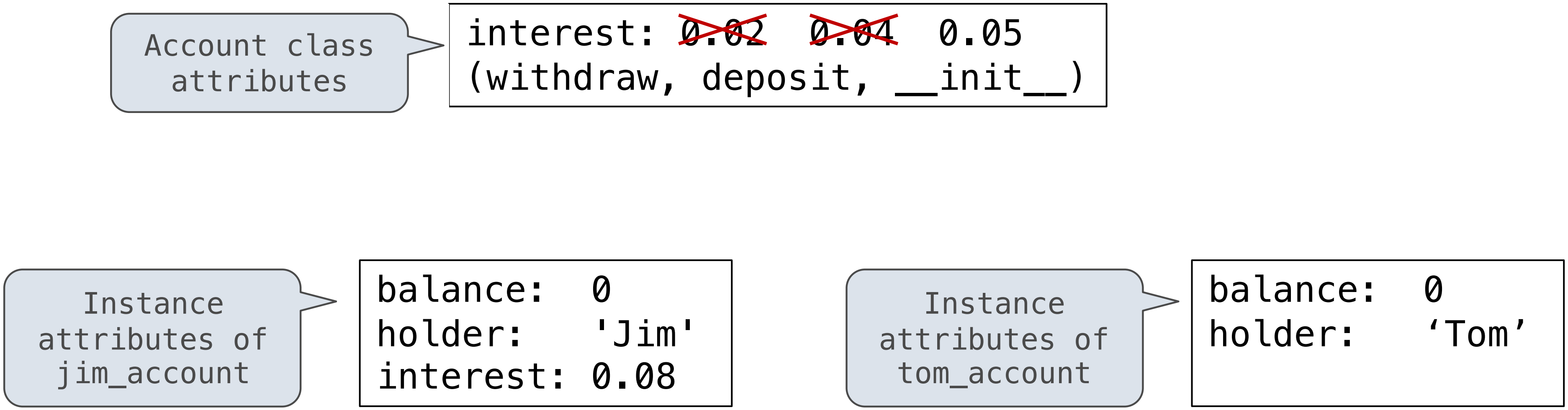
Instance attributes of tom_account

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
```

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

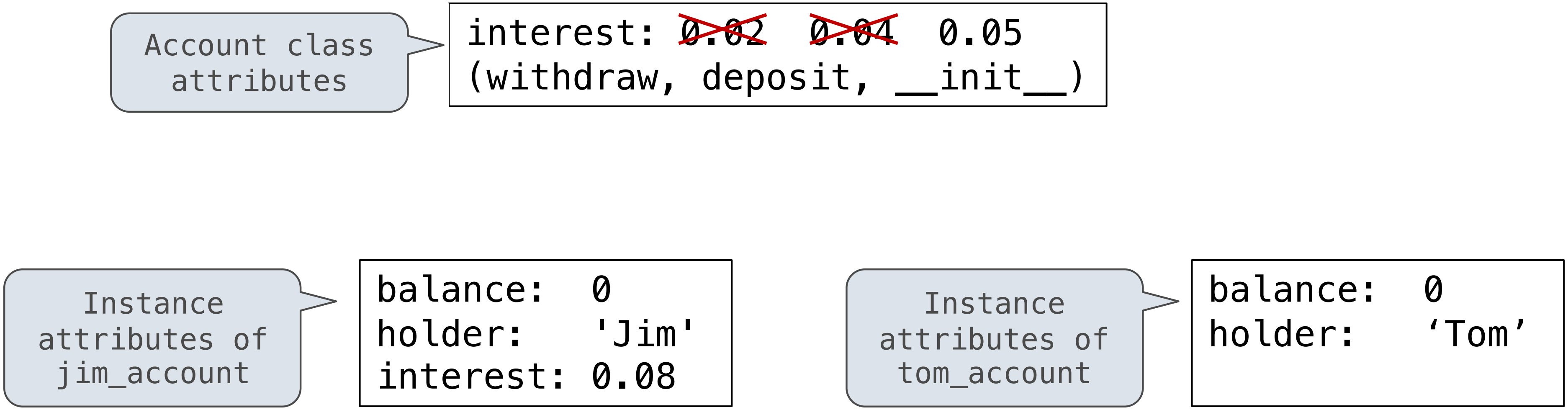
Instance attributes of tom_account

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
```

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

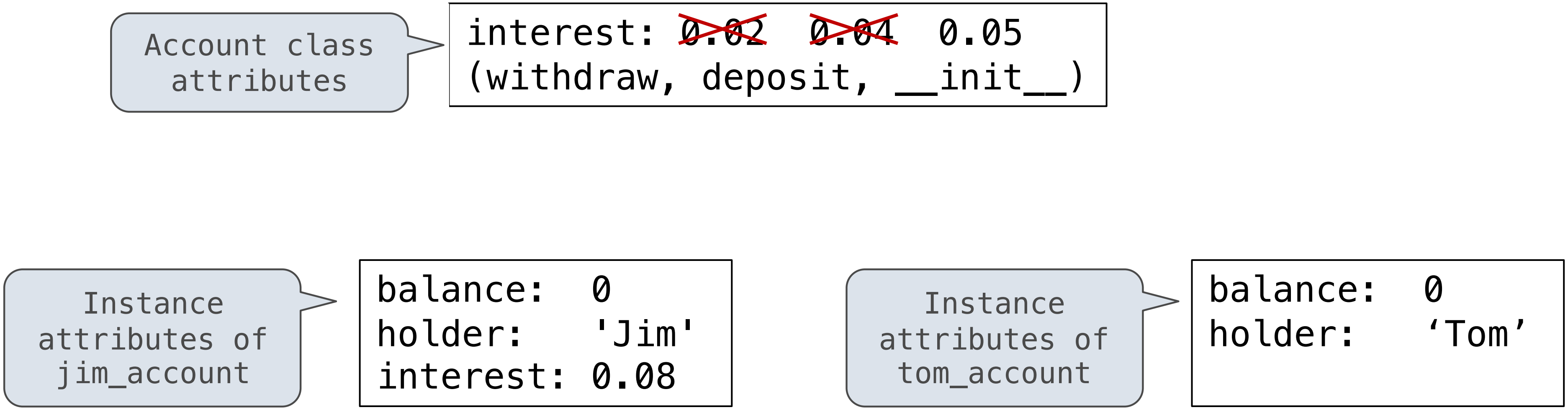
Instance attributes of tom_account

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
```

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
```

Lecture Overview

- Classes
  - Object Oriented Programming
  - Class Statements
  - Methods
  - Attributes
  - Inheritence

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)
```

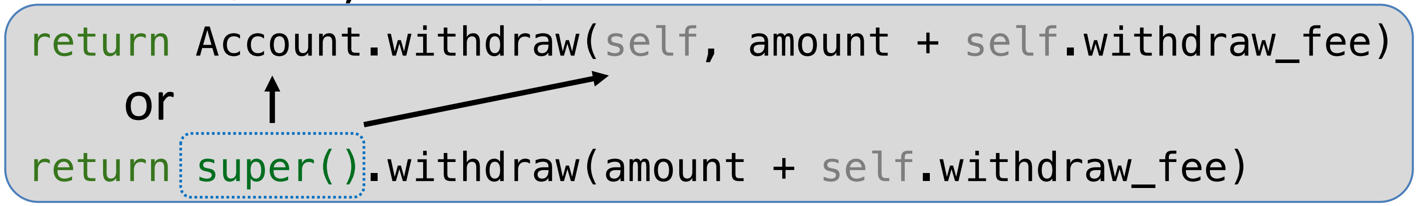
Inheritance Example

- 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)
```

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
```

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

Inheritance and Composition Example

- a bank **has a** collection of bank accounts it manages

```
class Bank:
    """A bank has accounts and pays interest."""
    def __init__(self):
        self.accounts = []

    def open_account(self, holder, amount, account_type=Account):
        """Open an account_type for holder and deposit amount."""
        account = account_type(holder)
        account.deposit(amount)
        self.accounts.append(account)
        return account

    def pay_interest(self):
        """Pay interest to all accounts."""
        for account in self.accounts:
            account.deposit(account.balance * account.interest)

    def too_big_too_fail(self):
          """Check whether the bank has more than 1 account or not."""
         return len(self.accounts) > 1
```

Inheritance and Composition Example

- a bank **has a** collection of bank accounts it manages

```
class Bank:
    """A bank has accounts and pays interest."""
    def __init__(self):
        self.account

    def open_account
        """Open an a
        account = ac
        account.depo
        self.account
        return accou

    def pay_interest
        """Pay inter
        for account
            account.

    def too_big_to_fail(self):
        """Check whether the bank has more than 1 account or not."""
        return len(self.accounts) > 1
```

```
>>> bank = Bank()
>>> john = bank.open_account('John', 10)
>>> jack = bank.open_account('Jack', 5, CheckingAccount)
>>> jack.interest
0.01
>>> john.interest = 0.06
>>> bank.pay_interest()
>>> john.balance
10.6
>>> jack.balance
5.05
>>> bank.too_big_to_fail()
True
```

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?

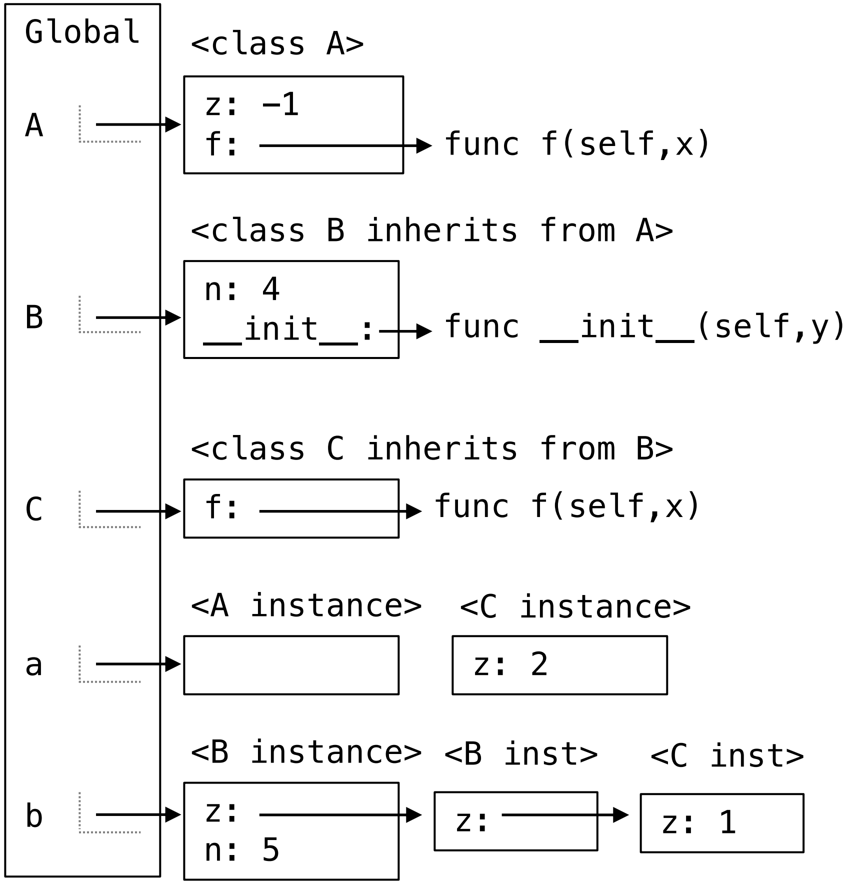
b.z

b.z.z

▷ b.z.z.z

b.z.z.z.z

None of these



Global

A

B

C

a

b

<class A>

z: -1

f: → func f(self,x)

<class B inherits from A>

n: 4

__init__: → func __init__(self,y)

<class C inherits from B>

f: → func f(self,x)

<A instance>

<C instance>

z: 2

<B instance>

z:

n: 5

<B inst>

z:

<C inst>

z: 1

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!
```

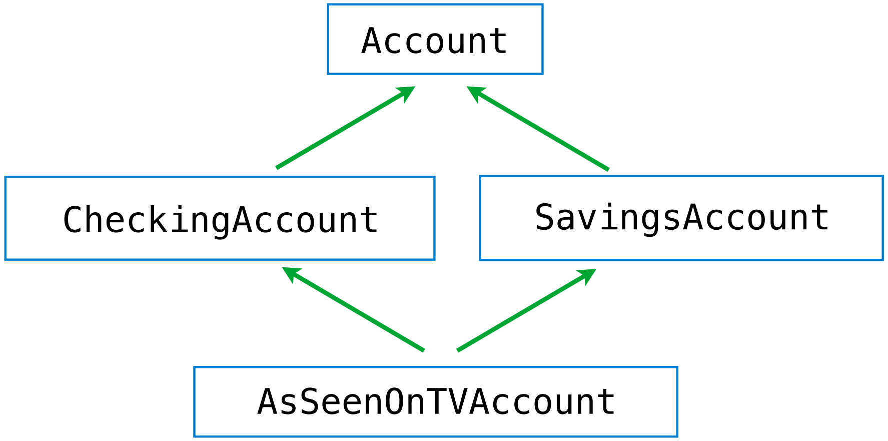
```
>>> such_a_deal = AsSeenOnTVAccount('John’)
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

Instance attribute

SavingsAccount method

CheckingAccount method

Resolving Ambiguous Class Attribute Names



Instance attribute

SavingsAccount method

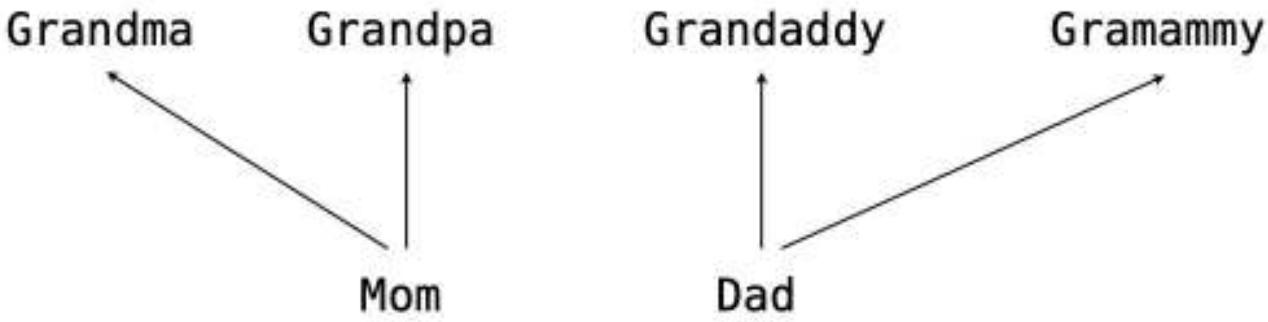
CheckingAccount method

```
>>> such_a_deal = AsSeenOnTVAccount('John’)
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

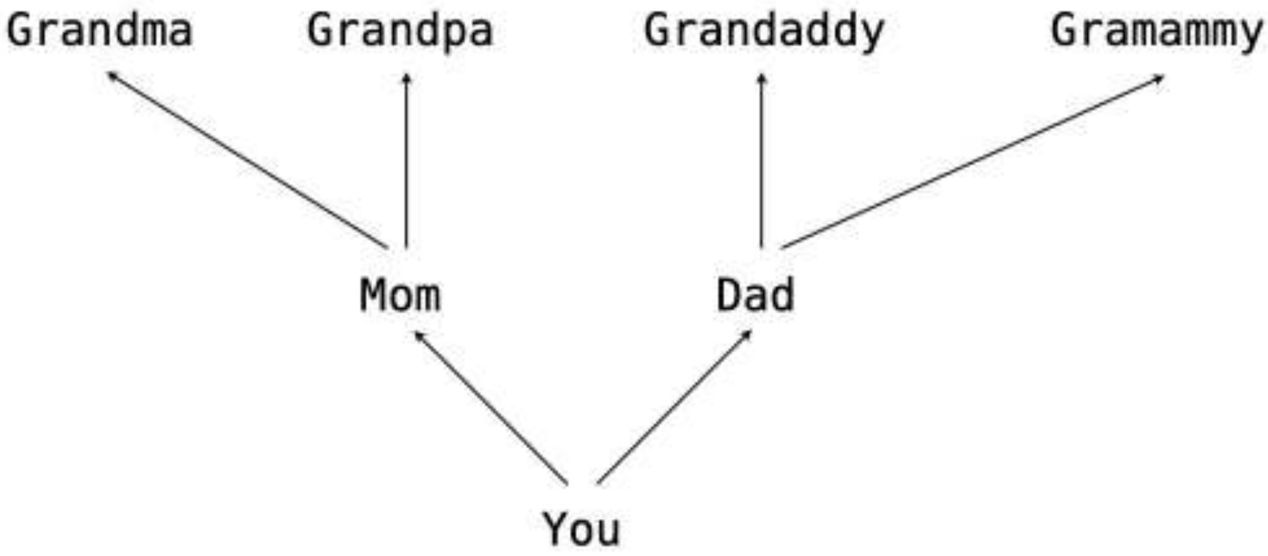
Biological Inheritance

Grandma Grandpa Grandaddy Gramammy

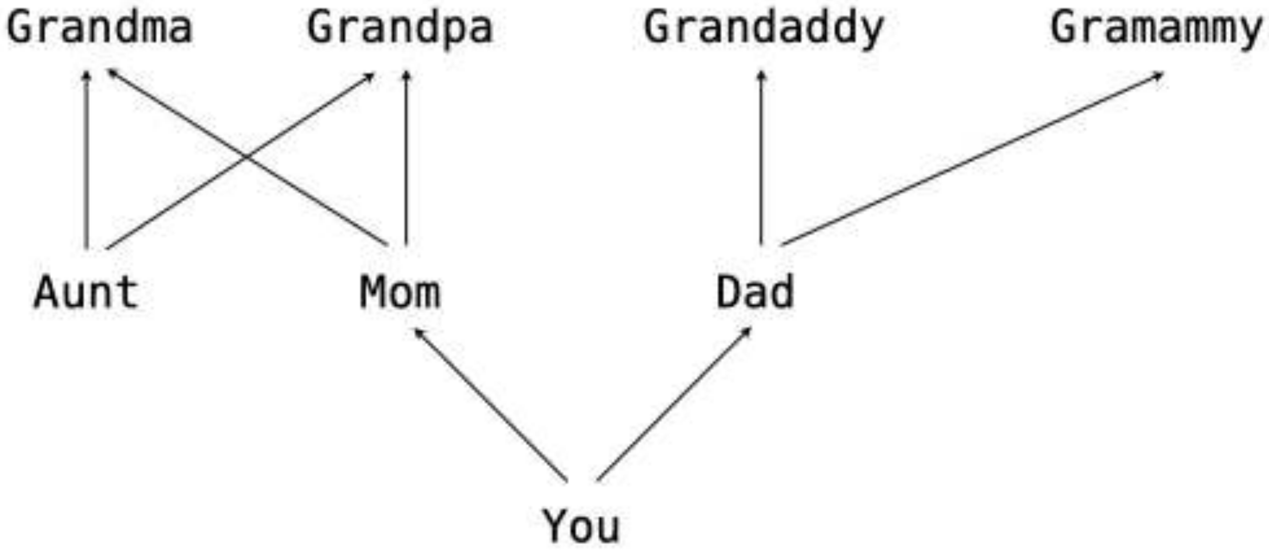
Biological Inheritance



Biological Inheritance



Biological Inheritance

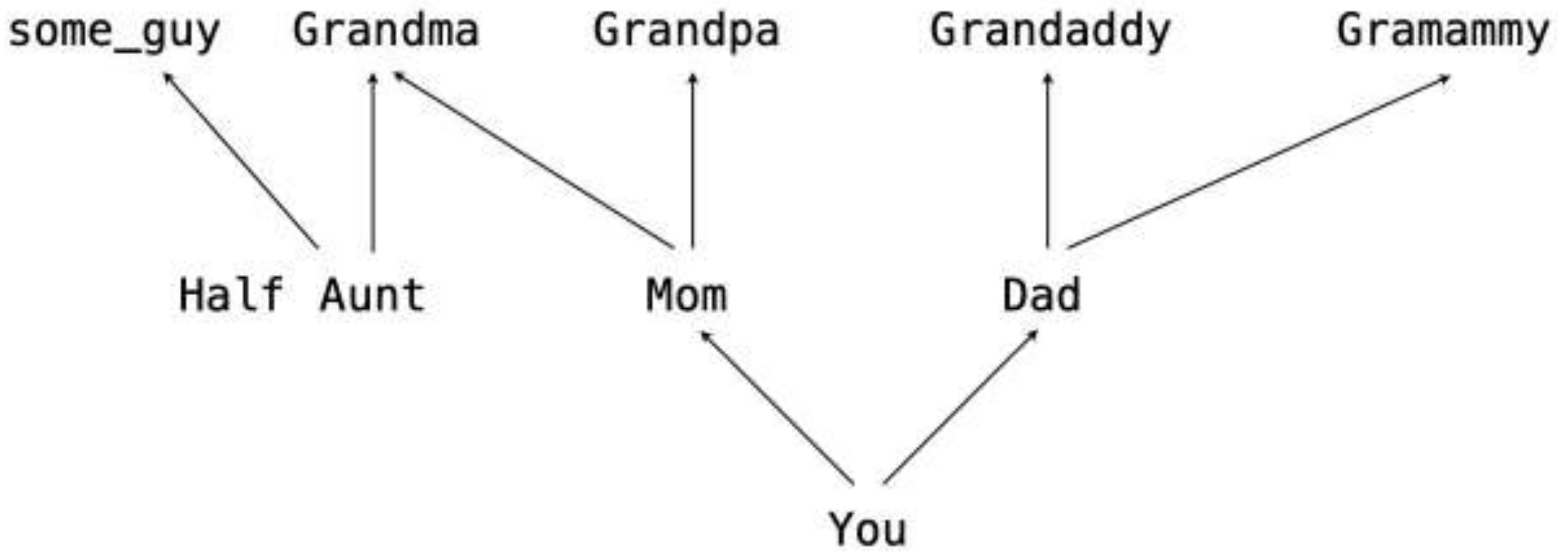


Grandma Grandpa Grandaddy Gramammy

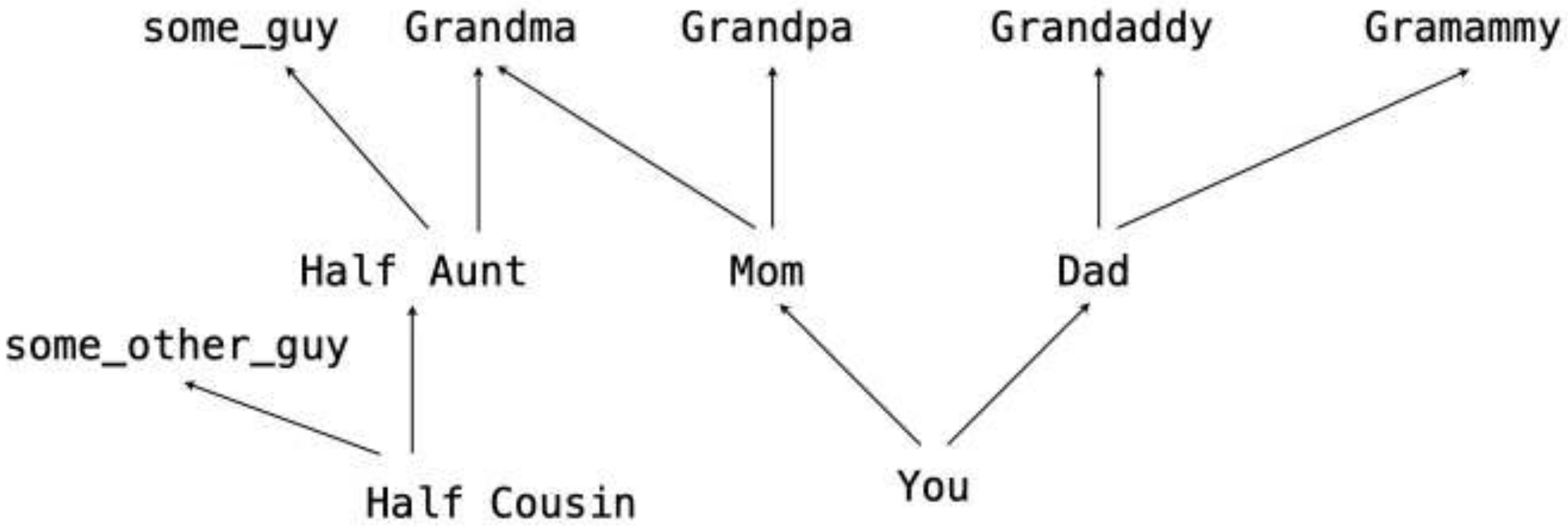
Aunt Mom Dad

You

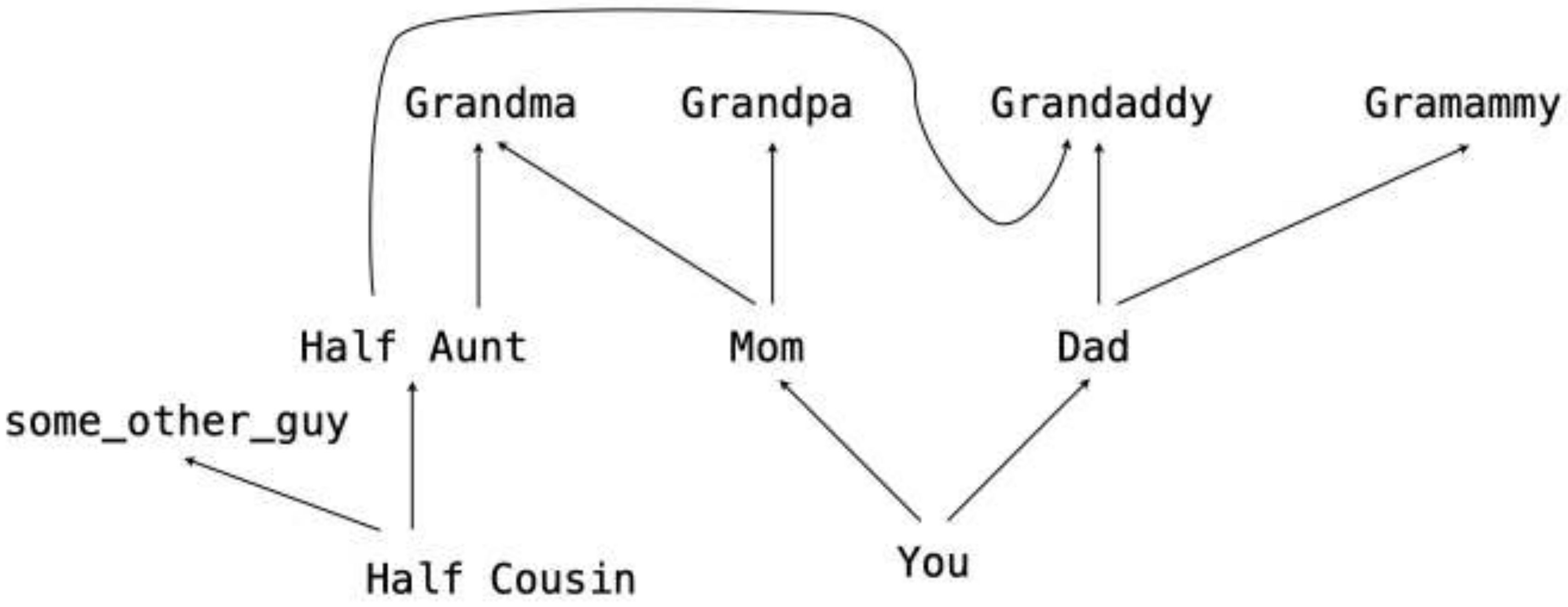
Biological Inheritance



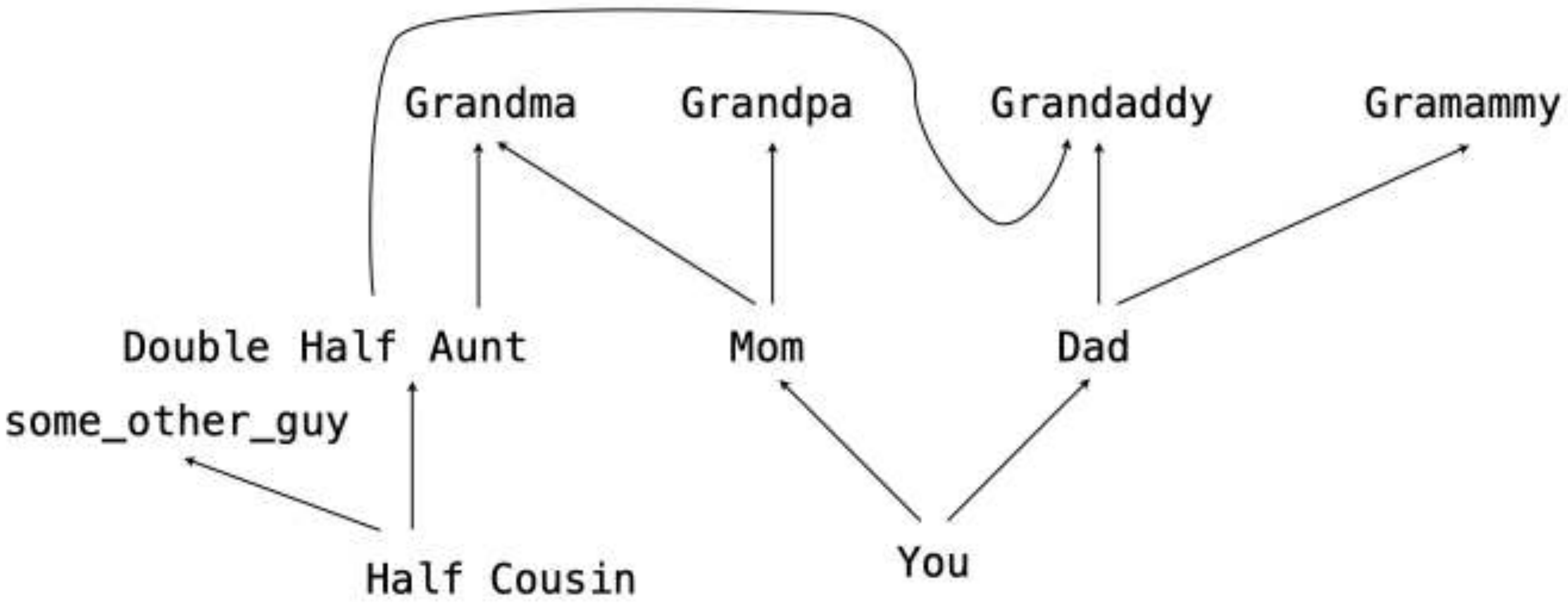
Biological Inheritance



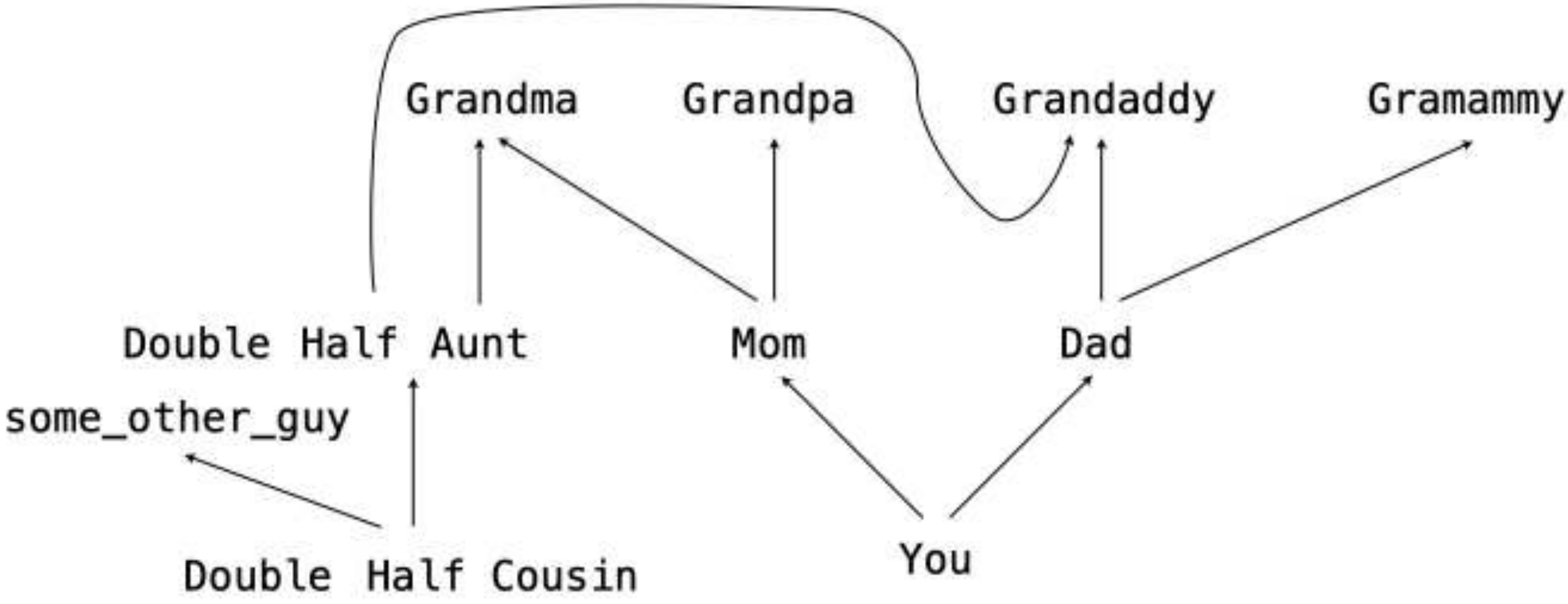
Biological Inheritance



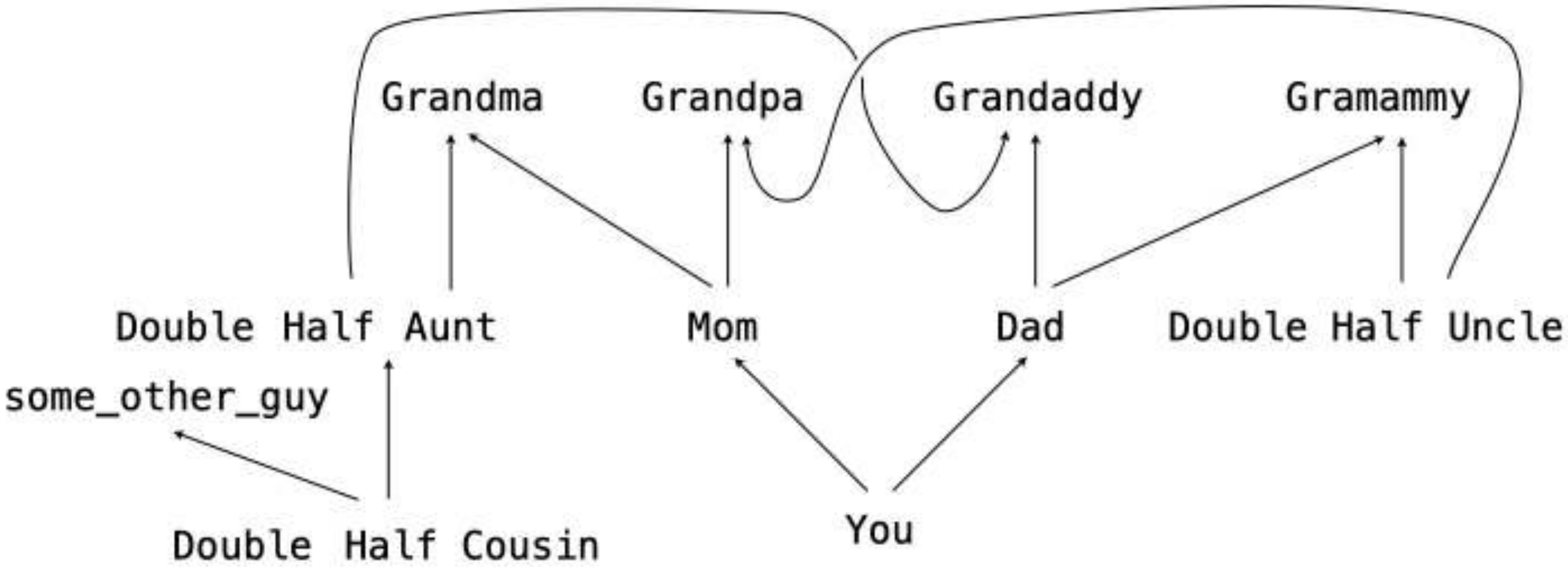
Biological Inheritance



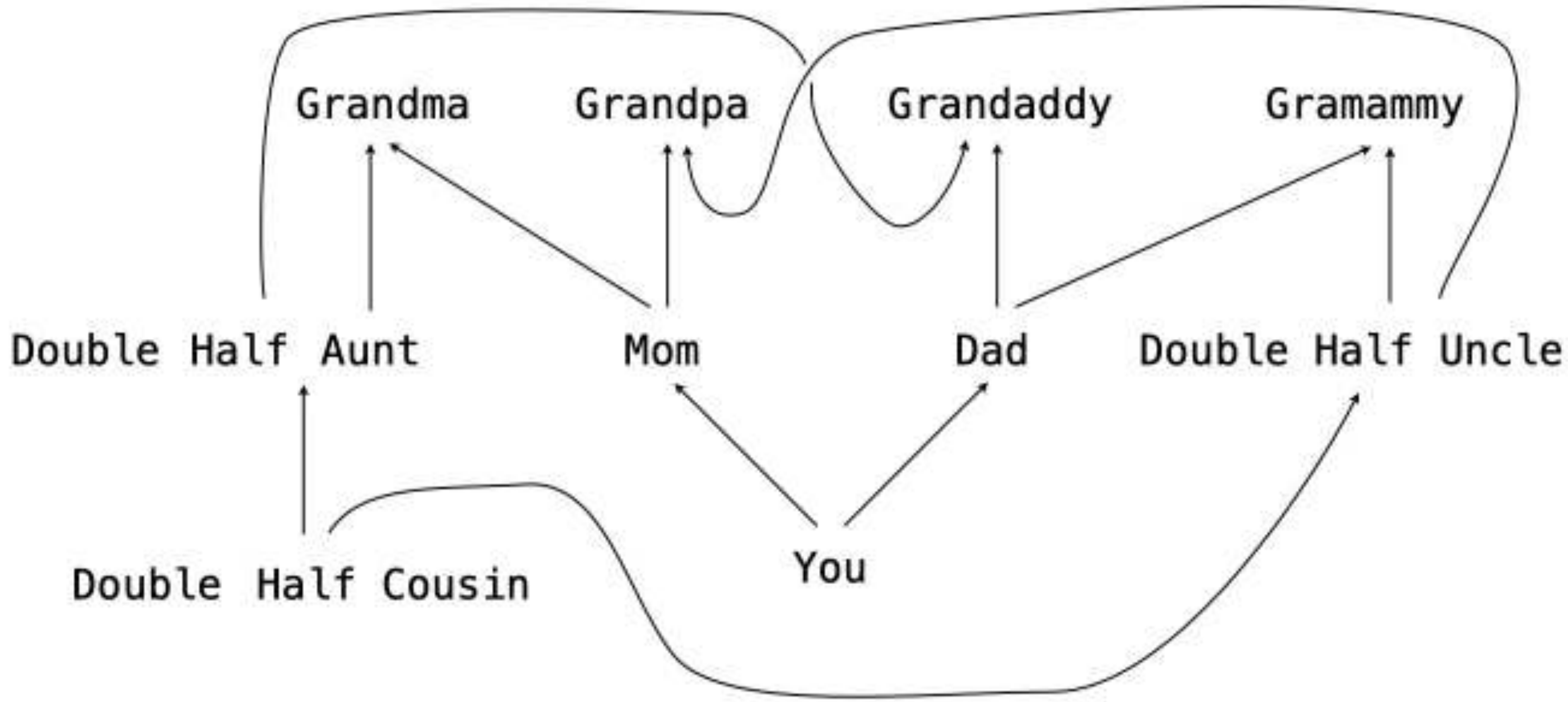
Biological Inheritance



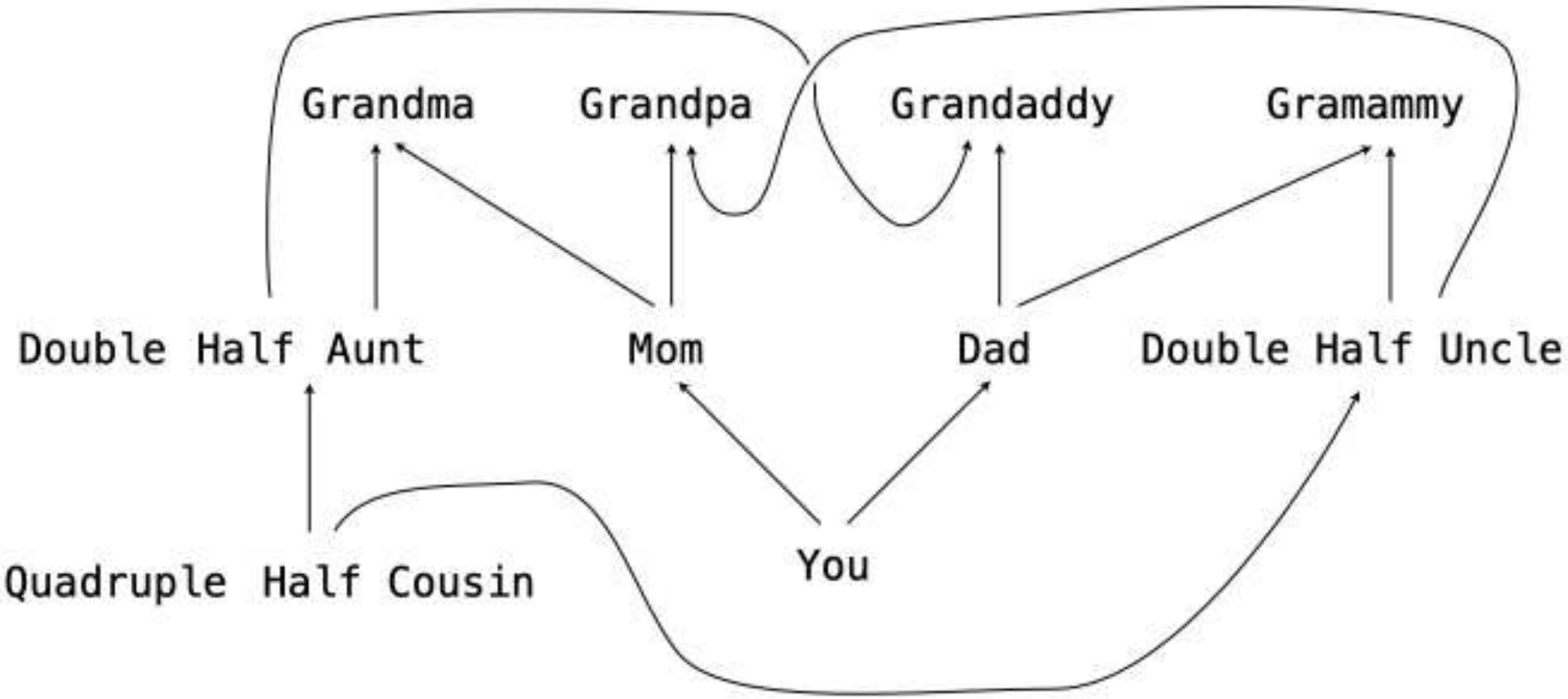
Biological Inheritance



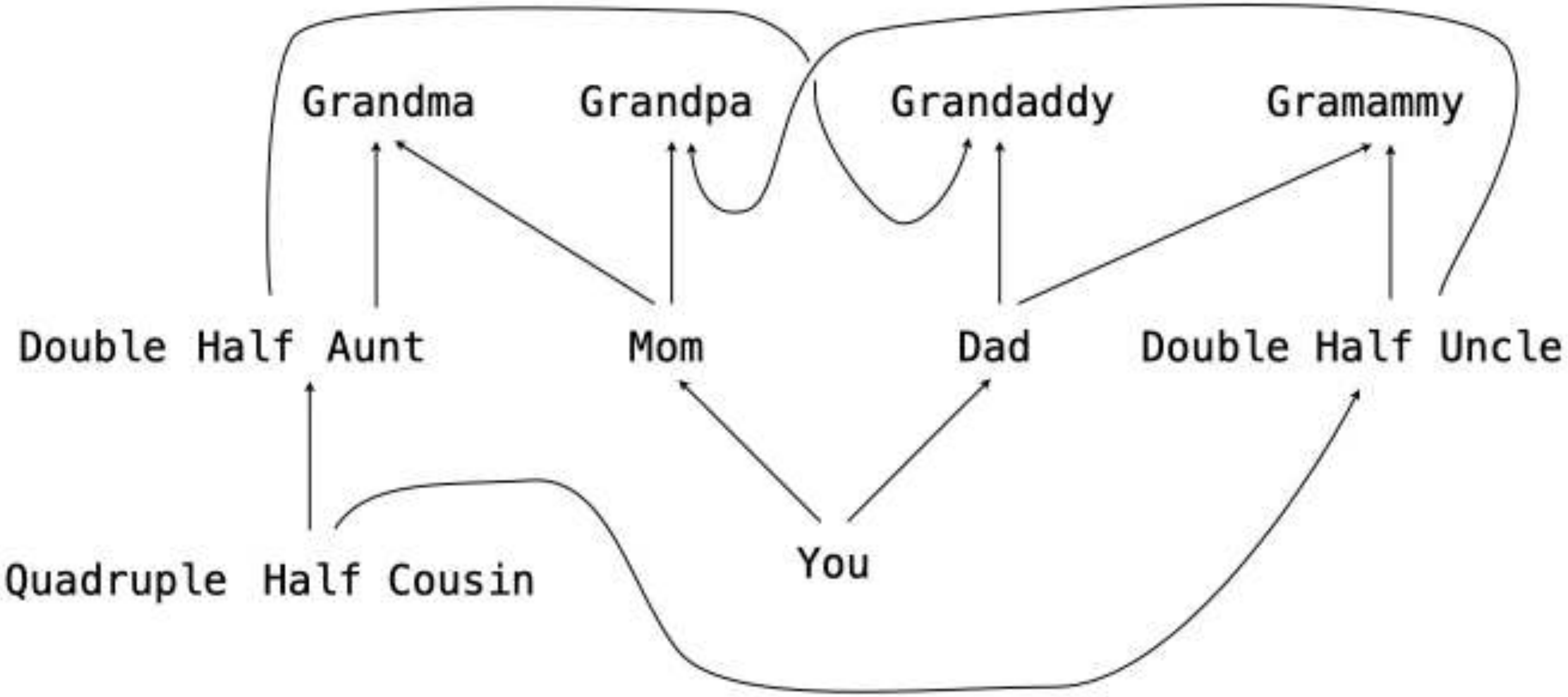
Biological Inheritance



Biological Inheritance



Biological Inheritance



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