Computational Thinking

Lecture 10: Class, Method, Subclass

University of Engineering and Technology
VIETNAM NATIONAL UNIVERSITY HANOI

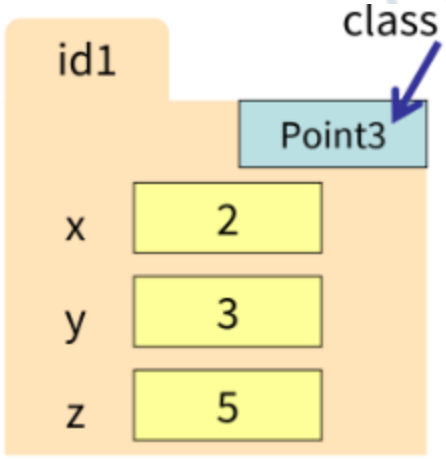
Outline

- Classes: initializers
- Methods and the self parameter
- Methods for string representations of objects
- Subclassing and inheritance
- The Python class hierarchy
- Method overriding
- Exceptions and subclassing
- Equivalence

Reading for today:
Think Python, 3rd ed.,
15.{1,2,5,6},
17.{1,2,5,8,9}

Classes: Constructors & Initializers

Classes



- An object's type is a **class**
- An object is an **instance** of a class
- The name of a class can be used as a **constructor function** that instantiates (creates) objects
- Classes also provide **methods**: special functions that can be used with objects

Example: a Class for 3D Points

```
class Point3:
    pass
```

The simplest possible class definition. Just a name, no body.

```
p = Point3()
```

Instantiate object of the class by calling constructor function. Same name as class.

```
p.x = 2
p.y = 3
p.z = 5
```

Assign to attributes.

Initializing Attributes

```
p = Point3()
p.x = 2
p.y = 3
p.z = 5

p2 = Point3()
p2.eks = 2
p2.z = 'five'
```

Tedious and error-prone to manually initialize attributes like this!

- What if we **forget** an attribute?
- What if we **misspell** an attribute name?
- What if we store data of **wrong type** in attribute?

We need a means to ensure attributes are initialized uniformly and correctly for all objects of the class.

The Initializer Method

two "double underscores" aka dunders

```
class Point3:
    def __init__(obj, x_val, y_val, z_val):
        obj.x = x_val
        obj.y = y_val
        obj.z = z_val

point1 = Point3(2, 3, 5)
```

Conventional parameter name: **self**

```
class Point3:
    def __init__(self, x_val, y_val, z_val):
        self.x = x_val
        self.y = y_val
        self.z = z_val

point1 = Point3(2, 3, 5)
```

Python programmers expect the first parameter name here to be **self**. Not `obj` or `this` etc. Python itself doesn't care which name you use, but for sake of good communication with other programmers, use **self**

Example: a Class for Social Media Posts

```
class Post:
    def __init__(self, text):
        """A post with whose text is `text`
        and with no likes so far."""
        self.text = text
        self.n_likes = 0

post1 = Post('good vibes')
```

A common convention: give the parameter and the attribute the same name.

Object Creation Protocol

You want to know all of this by heart.

Given a constructor function call:

`<class-name>(<arguments>)`

Python does the following:

1. Creates a **new object** (folder) in heap space
2. Gives that object an **identifier**
3. Calls the `__init__` method aka **initializer** of <class-name>
   A. Passes the new object's identifier as the **first** argument to the initializer
   B. Passes the remaining <arguments> (if any) from the constructor function call as the **rest** of the arguments to the initializer
   C. The initializer must return None (i.e., no return statement)
4. Returns the new object's identifier as the result of the constructor function call

Summary: Class Syntax So Far

Class Definition

```
class <class-name>:
    <method-definitions>
```

- The class name should be capitalized for new classes you create
- Built-in Python types (str, list, ...) do not always follow that rule

Initializer Method

```
def __init__(self, <args>):
    <body>
```

- The method name must be **`__init__`** with **dunders**
- The first argument should be named **`self`**

Do All Methods Have Dunder Names?

Definitely not! Recall these list methods:

lst.**index**(item)	The index of the first occurrence of item in lst.
lst.**insert**(i, item)	Insert item into lst just before the item at index i, shifting items to the right.
lst.**append**(item)	Insert item at the end of lst.
lst1.**extend**(lst2)	Append all the items of lst2 to lst1.

"Dunder methods" are for specialized Python-specific tasks.
Most methods we define will not be dunders.

Methods

What are Methods?

- Practically, **methods** are **functions** nested inside a class
- Conceptually, **methods** are **messages** we can send to objects
  - The object might exhibit or participate in some **behavior** in response to a message
  - The object might **reply** with a message (a return value); or it might not

```
class C:
    def __init__(self):
    ...
```

Example: "Messages" to a Tally Counter

- Tell me your **current count**
- **Increment** your count
- **Reset** your count

A Counter Class

```
class Counter:
    def __init__(self): ...
    def currCount(self): ...
    def incr(self): ...
    def reset(self): ...
```

```
>>> ctr = methods.Counter()
>>> ctr.currCount()
0
>>> ctr.incr()
>>> ctr.currCount()
1
```

How Methods Work

1. Methods use `self` to access attributes
2. Method calls make the object the `self` parameter
3. Methods are stored in class folder

1. Methods Use self to Access Attributes

```
class Counter:
    def __init__(self):
        self.count = 0

    def currCount(self):
        return self.count

    def incr(self):
        self.count = self.count + 1

    def reset(self):
        self.count = 0
```

Initializer creates an attribute in **self**'s folder

Other methods use that attribute through **self**

2. Method Calls Make the Object the self Param.

Method call syntax:

```
<object>.<method>(<args>)
```

The call is executed as:

```
<method>(<object>, <args>)
```

That is how the receiver object* becomes the value of the `self` parameter to the method

** the object that receives the message.*

2. Method Calls Make the Object the self Param.

Therefore if you forget the self parameter in a method header, you will get an error about a missing argument when that method is called.

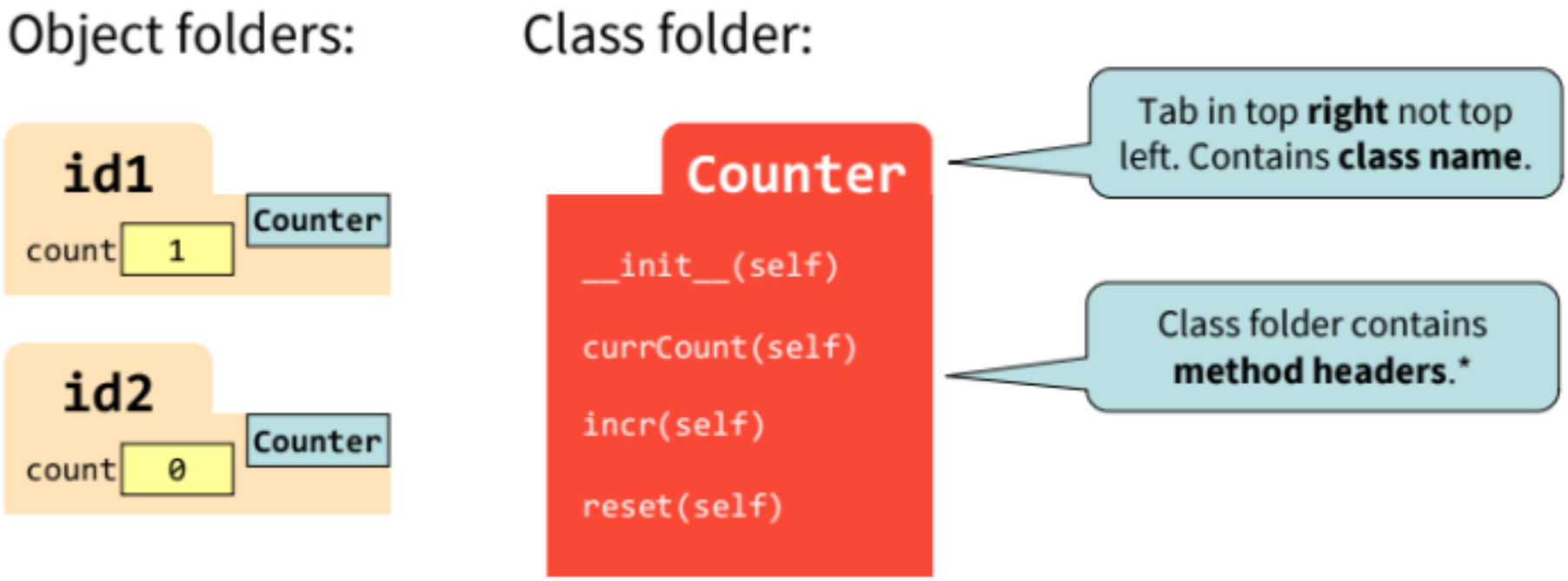
```
class Counter:
    def __init__(self):
        self.count = 0
    def currCount(self):
        return self.count
```

ctr.currCount()

currCount(ctr)

```
>>> ctr = Counter()
>>> ctr.currCount()
TypeError: Counter.currCount() takes 0 positional
arguments but 1 was given
```

3. Methods Are Stored in Class Folder



String Representations of Objects

Displaying Objects as Strings

```
>>> Point3(2, 3, 5)
<__main__.Point3 object at 0x10377e990>
>>> range(0, 10, 2)
range(0, 10, 2)
```

Q: How can we display our custom objects nicely to programmers in interactive mode?

A: By defining another method inside the class...

The String Representation Method

```
class Point3:
    def __repr__(self):
        return 'Point3(' \
            + str(self.x) + ', ' \
            + str(self.y) + ', ' \
            + str(self.z) \
            + ')'
```

```
>>> Point3(2, 3, 5)
Point3(2, 3, 5)
```

Converting Objects to Strings

```
>>> 'my int: ' + str(42)
'my int: 42'
# but that's not int(42)
>>> 'my point: ' + str(methods.Point3(2, 3, 5))
'my point: Point3(2, 3, 5)'
```

That is the result of __repr__() being called

Q: How can convert our custom objects to nice strings for non-programmers to read?

A: By defining yet another method inside the class...

The String Conversion Method

```
class Point3:
    def __str__(self):
        return '(' \
            + str(self.x) + ', ' \
            + str(self.y) + ', ' \
            + str(self.z) \
            + ')'
```

```
>>> str(Point3(2, 3, 5))
'(2, 3, 5)'
```

Summary: Objects To Strings

String Representation

- Define the __repr__() method
- Meant for **programmers** to read
- Main use: **display** in interactive mode (or in debugging messages)

String Conversion

- Define the __str__() method
- Meant for **non-programmers** to read
- Main use: **string argument** to str() or print()

With both, Python automatically calls the appropriate dunder method for us. We are customizing the behavior of Python by providing these special methods.

More Examples of Methods

Example: One Parameter

```
class Point3:
    def reflect(self):
        """Reflect this point through
        the origin."""
        self.x = -self.x
        self.y = -self.y
        self.z = -self.z
```

```
>>> p = Point3(1, 2, 3)
>>> p.reflect()
>>> p
Point3(-1, -2, -3)
```

Example: Two Parameters

```
class Point3:
    def scale(self, factor):
        """Scale this point by a numeric
        factor."""
        self.x = factor * self.x
        self.y = factor * self.y
        self.z = factor * self.z
```

```
>>> p = Point3(1, 2, 3)
>>> p.scale(2)
>>> p
Point3(2, 4, 6)
```

Example: Two Parameters, Both Points

```
class Point3:
    def distance(self, other):
        """Euclidean distance between
        two points."""
        xd = (other.x - self.x) ** 2
        yd = (other.y - self.y) ** 2
        zd = (other.z - self.z) ** 2
        return math.sqrt(xd + yd + zd)
```

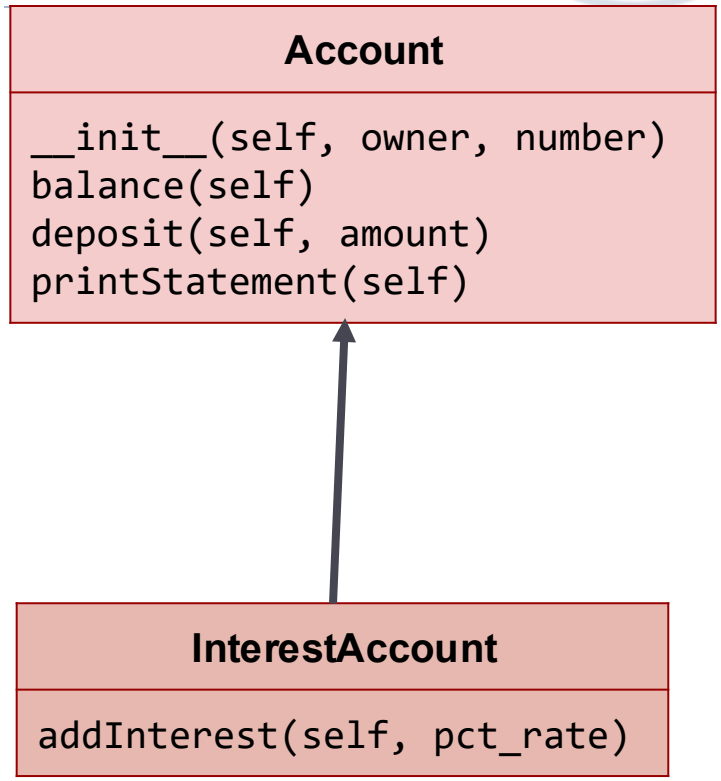
```
>>> p = Point3(1, 2, 3)
>>> p0 = Point3(0, 0, 0)
>>> p.distance(p0)
3.7416...
```

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

Subclassing and Inheritance

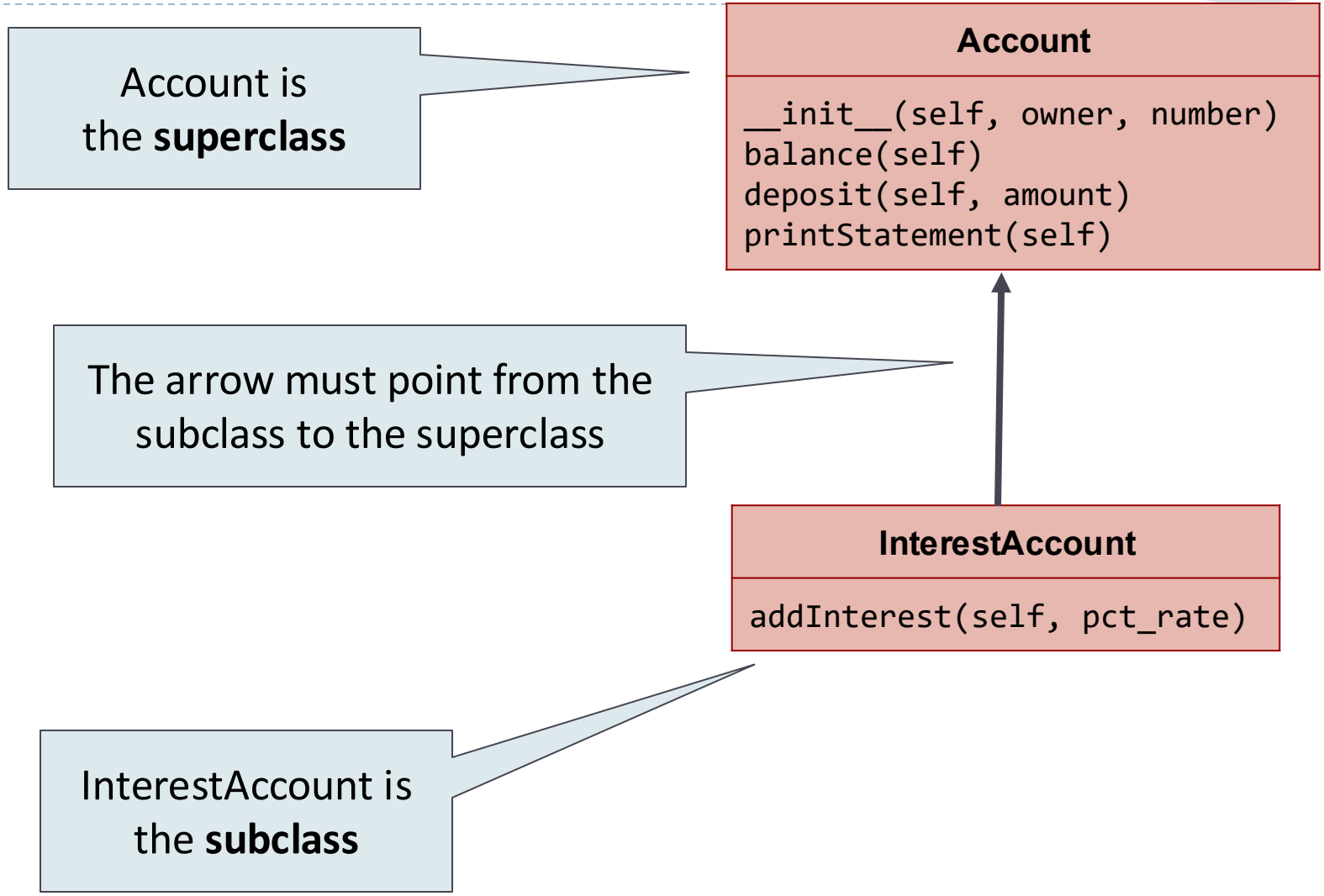
Subclassing

- A **subclass** is a class that is defined in terms of an already-existing class, which is called the **superclass**
- The subclass **specializes** aka **extends** the superclass by adding/customizing behaviors



Synonyms:
superclass = base class = parent class
subclass = derived class = child class

Subclass Diagrams



Subclass Definitions

Syntax

```
class <name>(<name>):
    <body>
```

The superclass is optional.

Example

```
class Account:
    ...
class InterestAccount(Account):
    ...
```

Meaning: The subclass **inherits** the methods of the superclass...

Subclass inherits the methods of the superclass...

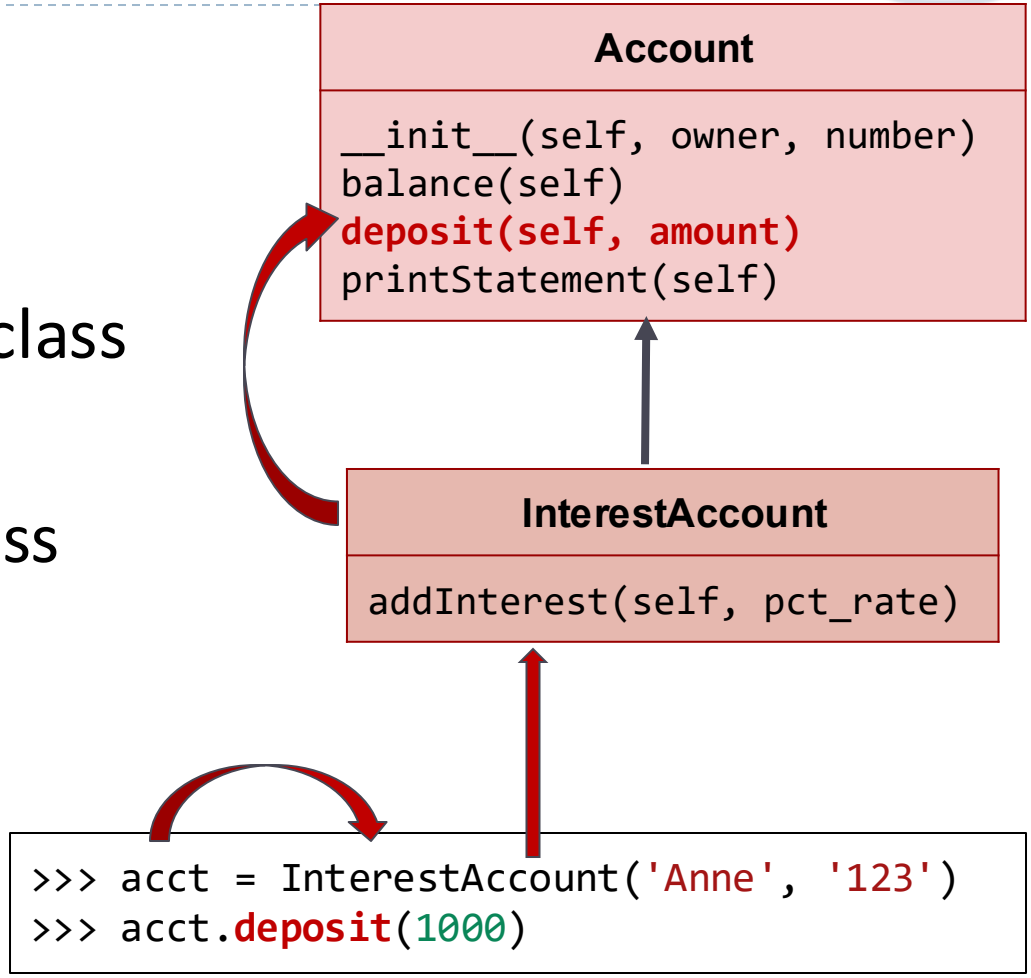
```
class Account:
    def __init__(self, owner, number):
        self.owner = owner
        self.number = number
        self.balance = 0
    def deposit(self, amount):
        self.balance += amount
    def printStatement(self):
        print('Owner: ' + self.owner)
        print('Account Number: ' + str(self.number))
        print('Balance: ' + str(self.balance))

class InterestAccount(Account):
    def addInterest(self, pct_rate):
        interest = self.balance * pct_rate
        self.deposit(interest)
```

```
>>> acct = InterestAccount('Anne', '123')
>>> acct.deposit(1000)
>>> acct.addInterest(0.05)
>>> acct.printStatement()
Owner: Anne
Account Number: 123
Balance: 1050.0
```

The Bottom-Up Rule for Finding Methods

To find a method:

- Start at the **bottom** by looking in the object's class
- If not there, look **up** to the superclass
- Keeping going **up** until found
- If never found, raise AttributeError



More Accounts, More Behaviors

Account

- Open account for owner
- Query the balance
- Make a deposit
- Print a statement

Interest-Bearing Account

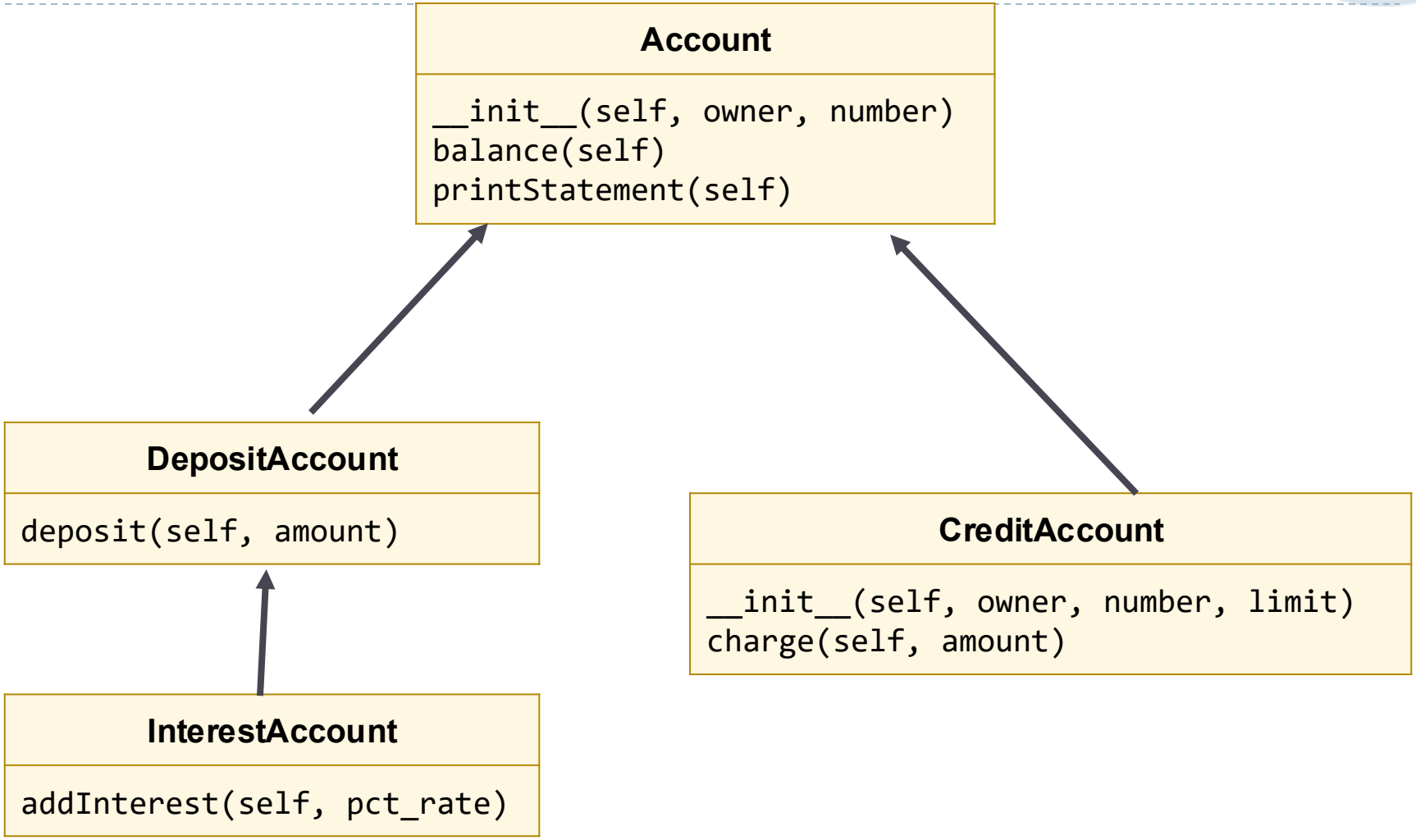
- base behaviours: Account
- Accrue interest

Credit Account

- What are base behaviors?
- Establish credit limit
- Make a charge, which might fail if over-draft

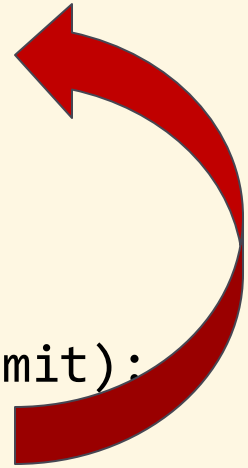
Query balance? YES
Make deposit? NO
Accrue interest? NO

Redesign of Account Classes



Initialization of Superclass Attributes

```
class Account:
    def __init__(self, owner, number):
        self._owner = owner
        self._number = number
        self._balance = 0.0

class CreditAccount(Account):
    def __init__(self, owner, number, limit):
        super().__init__(owner, number)
        self._limit = limit
```



super().<name>(<args>)
calls the method <name>
with **self** effectively as the receiver object
but starting the bottom-up-rule search in the superclass

Initialization of Superclass Attributes

```
class Account:
    def __init__(self, owner, number):
        self._owner = owner
        self._number = number
        self._balance = 0.0

class DepositAccount(Account):
    # No __init__() defined here

acct = DepositAccount('Anne', '2546154123')
```

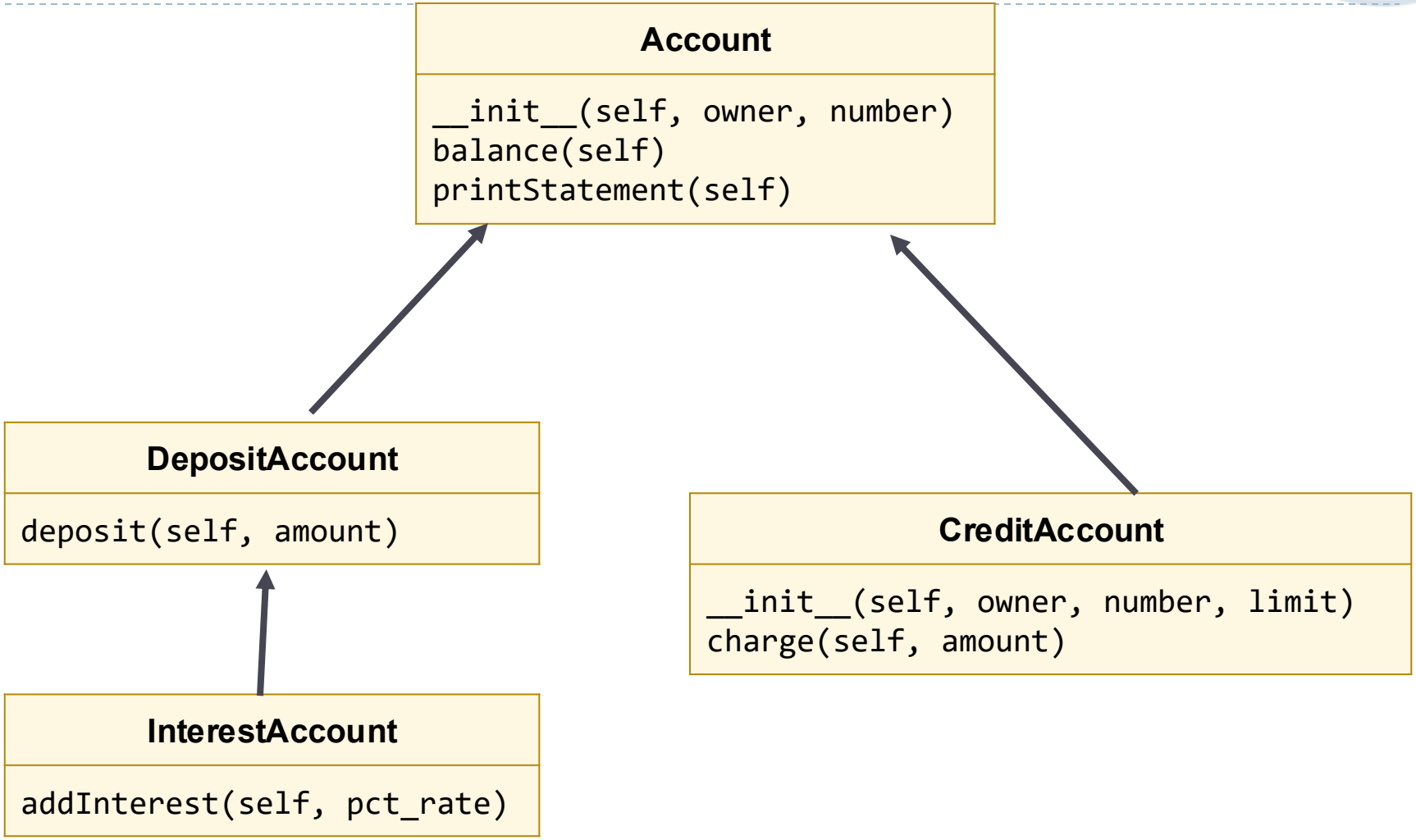
No initializer needed because DepositAccount does not have any state nor a class invariant of its own.

The bottom-up rule means that Account's __init__() will be called when the DepositAccount constructor function is called:

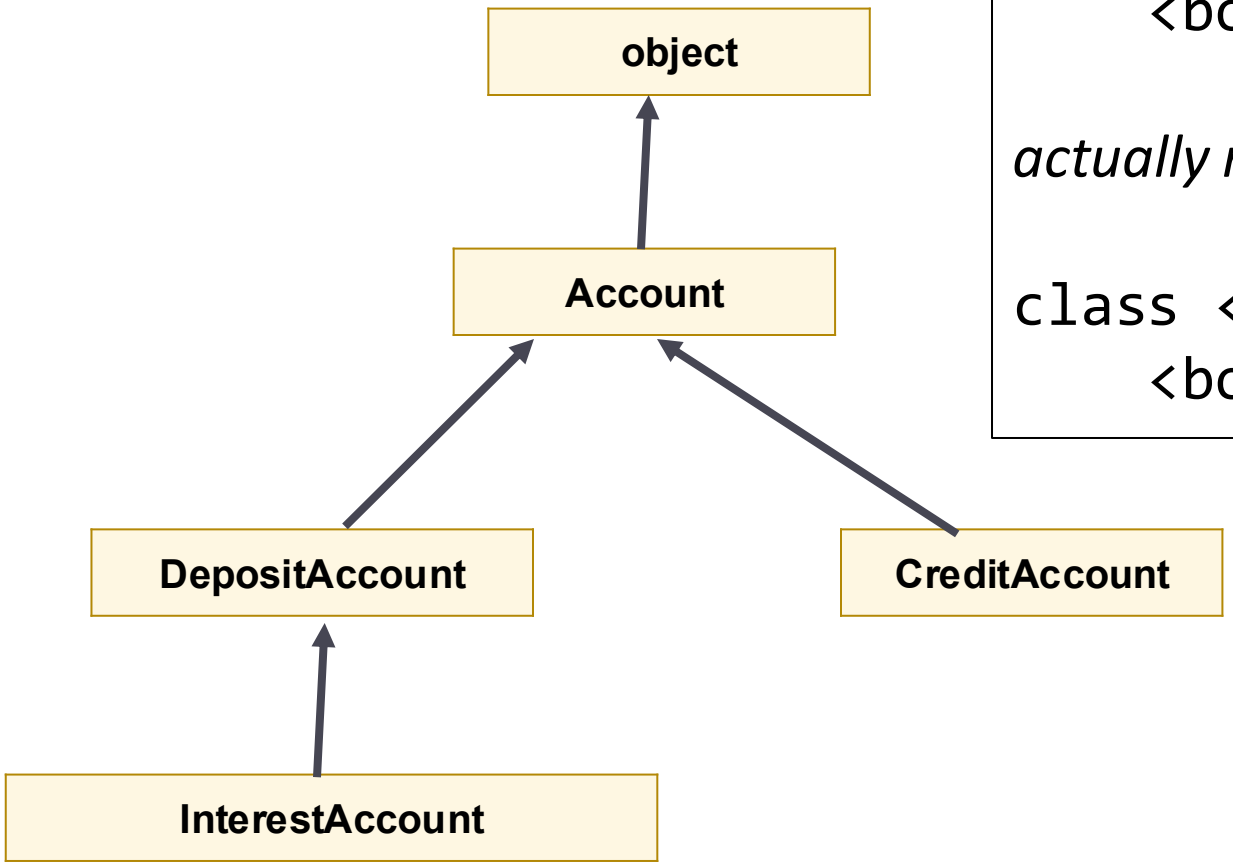
- Start searching for __init__() at bottom in DepositAccount. Not found.
- Go up to superclass Account. Found.

The Python Class Hierarchy

Subclassing Creates a Class Hierarchy



The Super-est Class of All is **object**




```
class <name>:
    <body>

actually means

class <name>(object):
    <body>
```

The **object** Class

```
# effectively built-in to Python
class object:
    ...
    def __str__():
        """returns '<class-name> object at <address>'"""
```

```
# our own code
class C(object):
    ...
    # no __str__() method defined
```

```
>>> c = C()
>>> str(c)
```

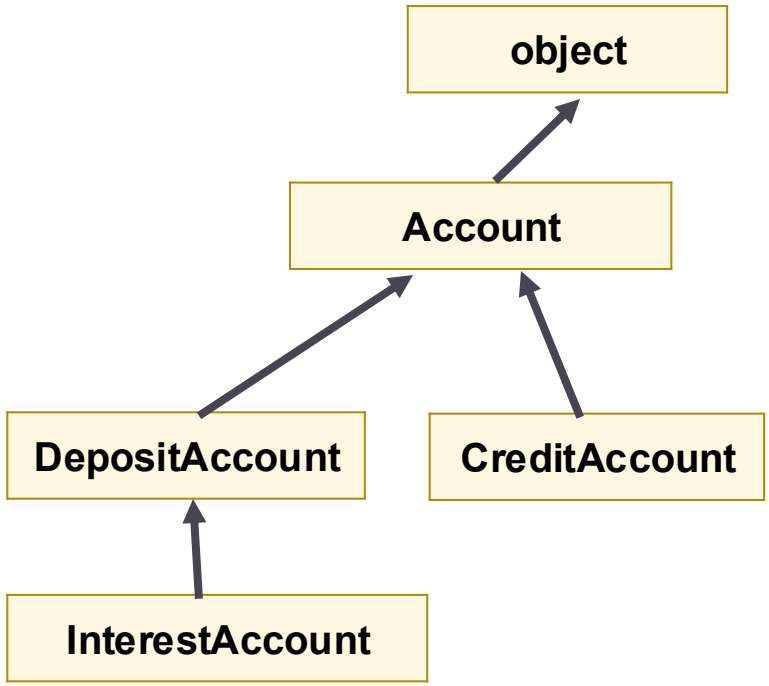
Built-in function `str(x)` effectively calls `x.__str__()`

The bottom-up rule means that `object`'s `__str__()` is called:

- Start searching for `__str__()` at bottom in C. Not found.
- Go up to superclass `object`. Found.

Testing Types

```
>>> a = InterestAccount('Bean', '274563784')
>>> isinstance(a, InterestAccount)
True
>>> isinstance(a, DepositAccount)
True
>>> isinstance(a, Account)
True
>>> isinstance(a, object)
True
>>> isinstance(a, CreditAccount)
False
>>> type(a) == InterestAccount
True
>>> type(a) == Account
False
```



Read → as "is a"

Testing Types

`isinstance()`

- Returns a **Boolean**
- An object is an instance of its class and
  all its superclasses and
  all their superclasses, etc.

`type()`

- Returns an **object** representing a class
- An object has but **one** type

Method Overriding

Overriding

When a subclass defines a method already defined by a superclass we say the method is **overridden**: the definition in the subclass takes precedence according to the bottom-up rule

Example 1: Overrides `Account.printStatement()`

```
class Account:
    def printStatement(self):
        print('Statem
        print('Accoun
        print('Owner:
        print('Balanc
```

```
class CreditAccount(Account):
    def printStatement(self):
        print('Statement:')
        print('Account #: ' + self._number)
        print('Owner: ' + self._owner)
        print('Balance: $' + str(self._balance))
        print('Limit: $' + str(self._limit))
```

Overriding

Example 1: Overrides `Account.printStatement()`

```
class Account:
    def printStatement(self):
        print('Statem
        print('Accoun
        print('Owner:
        print('Balanc
```

```
class CreditAccount(Account):
    def printStatement(self):
        super().printStatement()
        print('Limit: $' + str(self._limit))
```

Calling the superclass's version of the method means we avoid copying code

Review: super().<name>(<args>) calls the method <name> with self effectively as the receiver object but starting the bottom-up-rule search in the superclass.
So here the search starts in the superclass of CreditAccount, which is Account.

Overriding

When a subclass defines a method already defined by a superclass we say the method is **overridden**: the definition in the subclass takes precedence according to the bottom-up rule

Example 2: Overrides `object.__str__()`

```
class object:
    def __str__(self):
        #....
```

```
class Account:
    def __str__(self):
        return 'Account #' + self._number
```

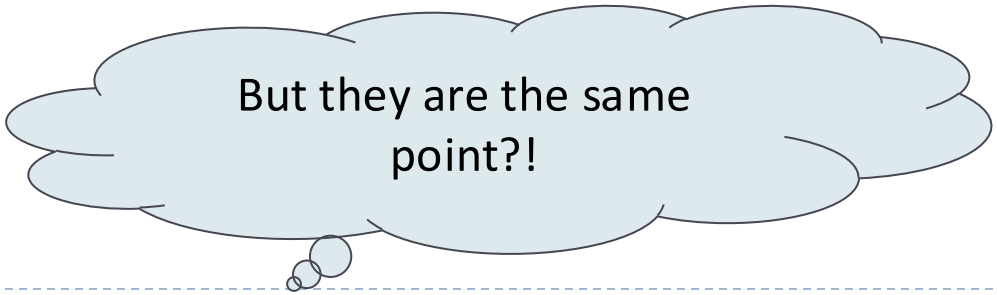
Equivalence

What does `==` mean?

```
>>> 42 == 42
True
```

```
class Point3:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

```
>>> Point3(0,0,0) == Point3(0,0,0) # 2 objects
False
```



Equality: (Pythonically) It's Complicated

Equality: Same Object

- The `is` operator
- Do the two objects have the same identifier (i.e. same memory address), thus they are the same — guaranteed identical?

Equivalence: Alike Objects

- The == operator
- Are the two objects alike even if not the same — not identical?
- Through **overriding**, programmers can customize `==` for each class to focus on what they consider to be important differences
- Python already has such customization even with int and str objects...

Equivalence with a Built-in Class

```
>>> def incr(n): return n + 1
>>> x = 1000          # *
>>> y = incr(x) - 1
>>> x == y
True
>>> x is y
False
```

Python customizes`==` for int to ensure integer equivalence even if different objects

* Depending on Python implementation, different starting values of x might or might not exhibit the behavior shown here. Likewise, similar but simpler code might or might not. The implementation is free to intern objects to save memory: instead of creating a new object, it re-uses an existing object. When that happens, the two operators will agree.

Equivalence with Custom Classes

To define our own notion of equivalence for a class, we can override a special dunder method: `__eq__()`

```
class Point3:
    ...
    def __eq__(self, other):
        """Same coordinates"""
        return type(other) == Point3 \
            and self.x == other.x \
            and self.y == other.y \
            and self.z == other.z
```

Why it works:
`<expr1> == <expr2>`
is translated* to:
`<expr1>.__eq__(<expr2>)`

```
>>> Point3(0,0,0) == Point3(0,0,0)
True
```

* But instead to `<expr2>.__eq__(<expr1>)` if `<expr2>`'s class is a subclass of `<expr1>`'s class, so that the more specific method is called.

The Implementation of Point3.__eq__()

Two parameters:
LHS and RHS of ==

Check that RHS is a Point3 object* before attempting to access its attributes

```
class Point3:
    ...
    def __eq__(self, other):
        """Same coordinates"""
        return type(other) == Point3 \
            and self.x == other.x \
            and self.y == other.y \
            and self.z == other.z
```

Why it works:
`<expr1> == <expr2>`
is translated* to:
`<expr1>.__eq__(<expr2>)`

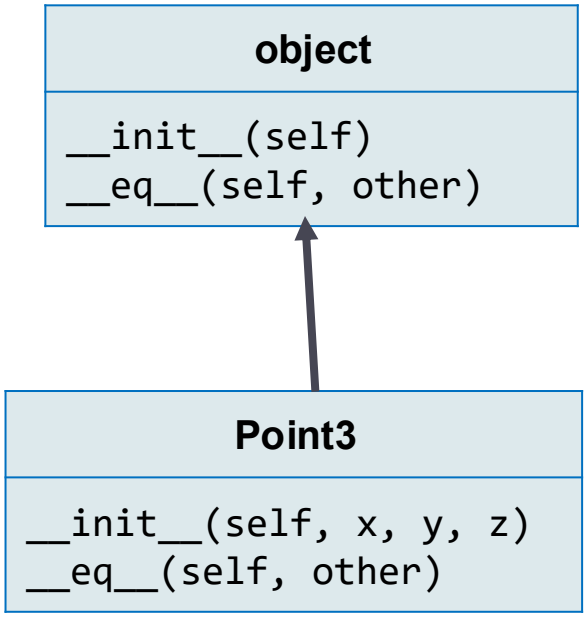
Check that LHS and RHS have the same x,y,z coordinates

* Why type() not instanceof()? We choose here to consider Point3 objects equivalent only to other Point3 objects, not to objects of classes that might later subclass Point3.

Overriding __eq__()

The object class defines __eq__() essentially as:

```
class object:
    ...
    def __eq__(self, other):
        return (self is other)
```



So by default, programmer-defined classes implement == the same as is: equality and equivalence are the same by default. That explains the original, uncustomized behavior of Point3.

Other Operators are Also Overrides

Operator	Method*
==	__eq__
!=	__ne__
+	__add__
-	__sub__
and	__and__
or	__or__
<	__lt__
<=	__le__
>	__gt__
>=	__ge__
many others	

* We don't expect you to memorize these method names, except for __eq__.

Summary

- A *class* defines a blueprint, while an *object* is an instance created via the constructor __init__.
- Methods belong to objects, and self refers to the instance receiving the method call.
- Special (dunder) methods like __repr__ and __str__ customize object behavior and representation.
- Subclasses inherit attributes and methods from superclasses to enable code reuse and specialization.
- Method lookup follows a bottom-up order, from the subclass to its parent classes.
- super() allows subclasses to extend, not duplicate, superclass behavior