CS 1110:
Introduction to Computing Using Python

Lecture 17

Classes

[Andersen, Gries, Lee, Marschner, Van Loan, White]

Announcements

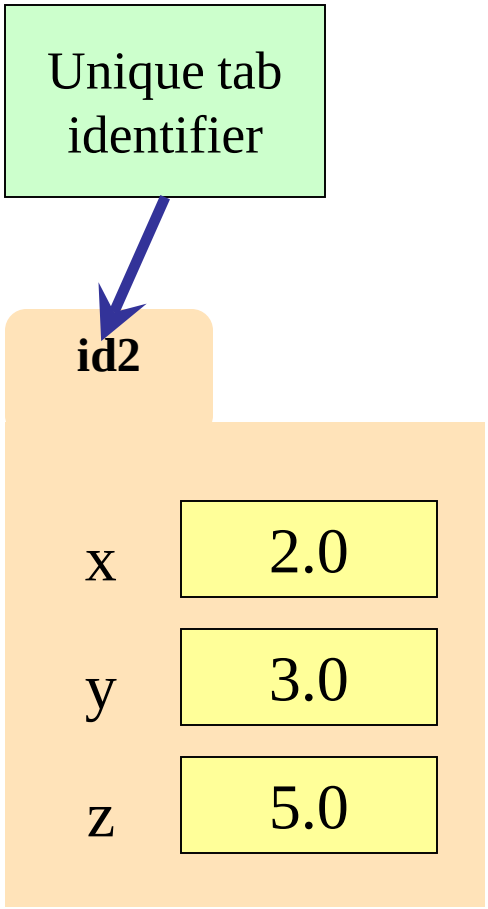
- Lab 9 is out. Due in two weeks because of break.
- Prelim 1 solutions posted on Exams page.
- Regrade request instructions have been emailed.
- Makeup exams are in homework handback room.
- A3 due Thursday.
- No A4 until after break!

Announcements

- For more solved recursion examples, see the demos:
  - http://www.cs.cornell.edu/courses/cs1110/2014sp/lectures/index.php
  - https://www.cs.cornell.edu/courses/cs1110/2016fa/lectures/10-18-16/modules/morefun.py
- For more daily practice: download the demo code we post for each lecture; remove the contents, and try to reproduce what we did in class.

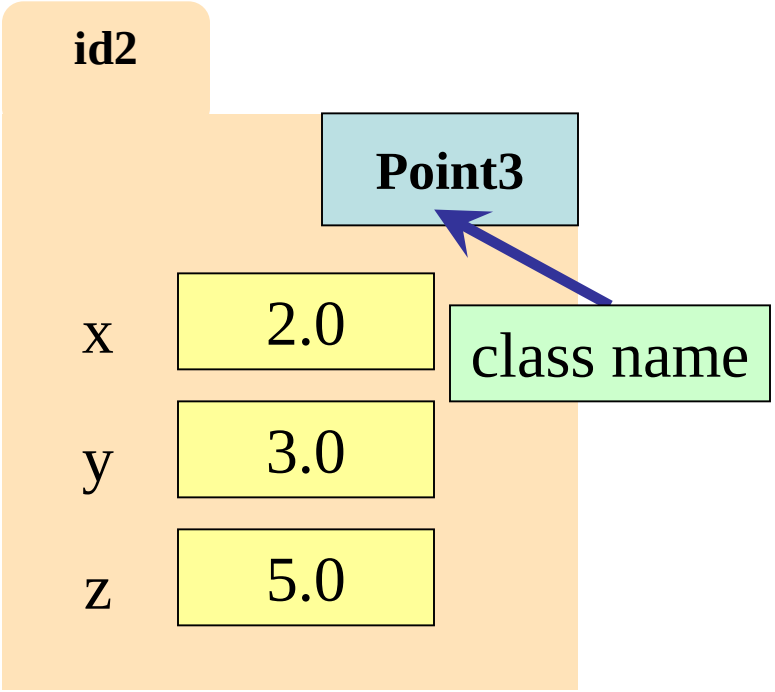
Recall: Objects as Data in Folders

- An object is like a **manila folder**
- It contains other variables
  - Variables are called **attributes**
  - Can change values of an attribute (with assignment statements)
- It has a "tab" that identifies it
  - Unique number assigned by Python
  - Fixed for lifetime of the object

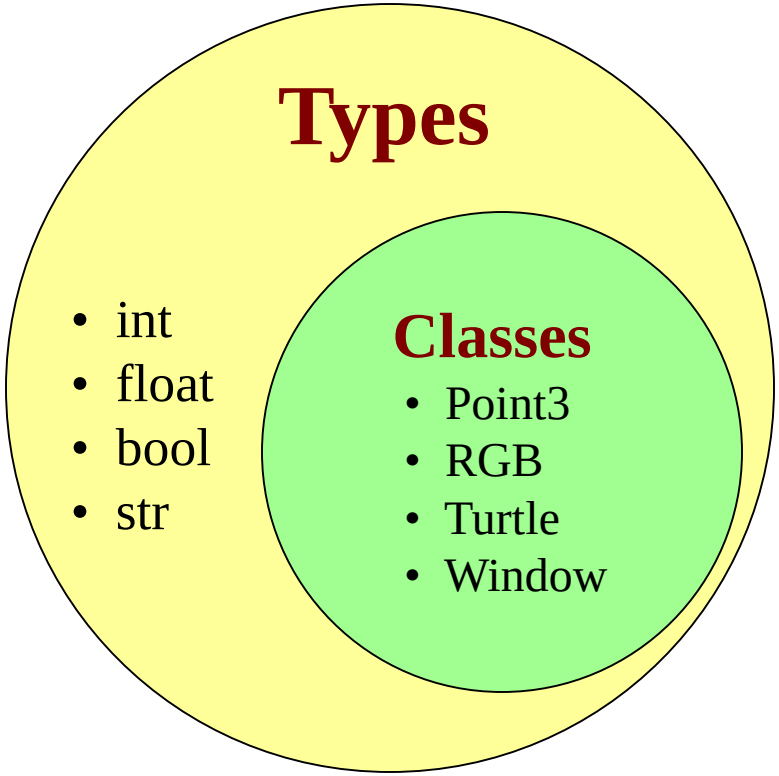


Recall: Classes are Types for Objects

- Values must have a type
  - An object is a **value**
  - Object type is a **class**



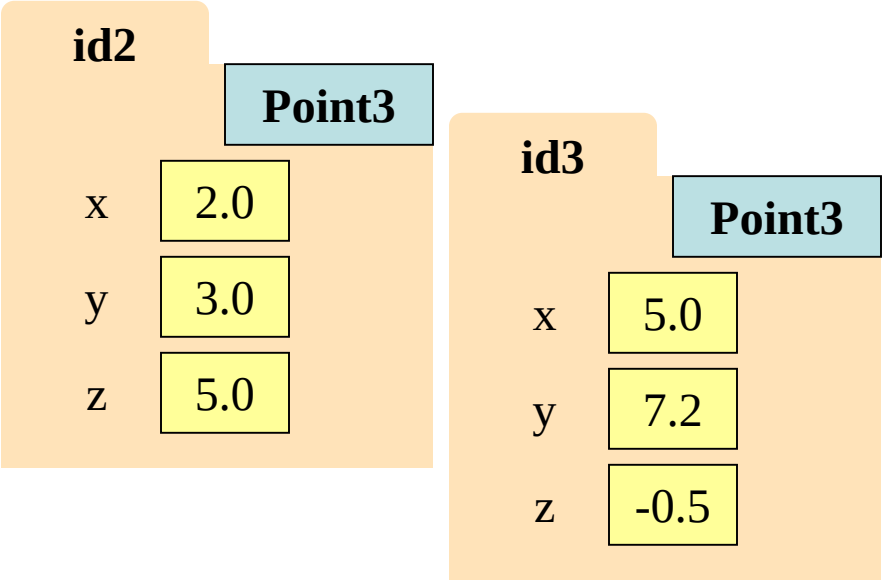
- Classes are how we add new types to Python



Classes Have Folders Too

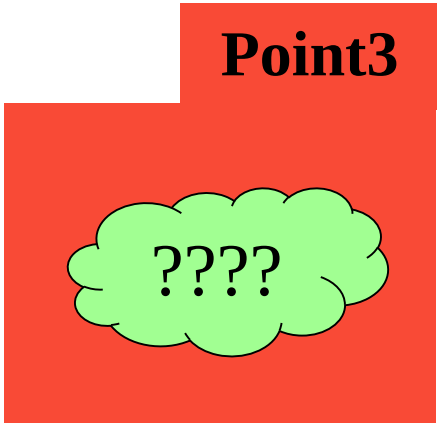
Object Folders

- Separate for each *instance*



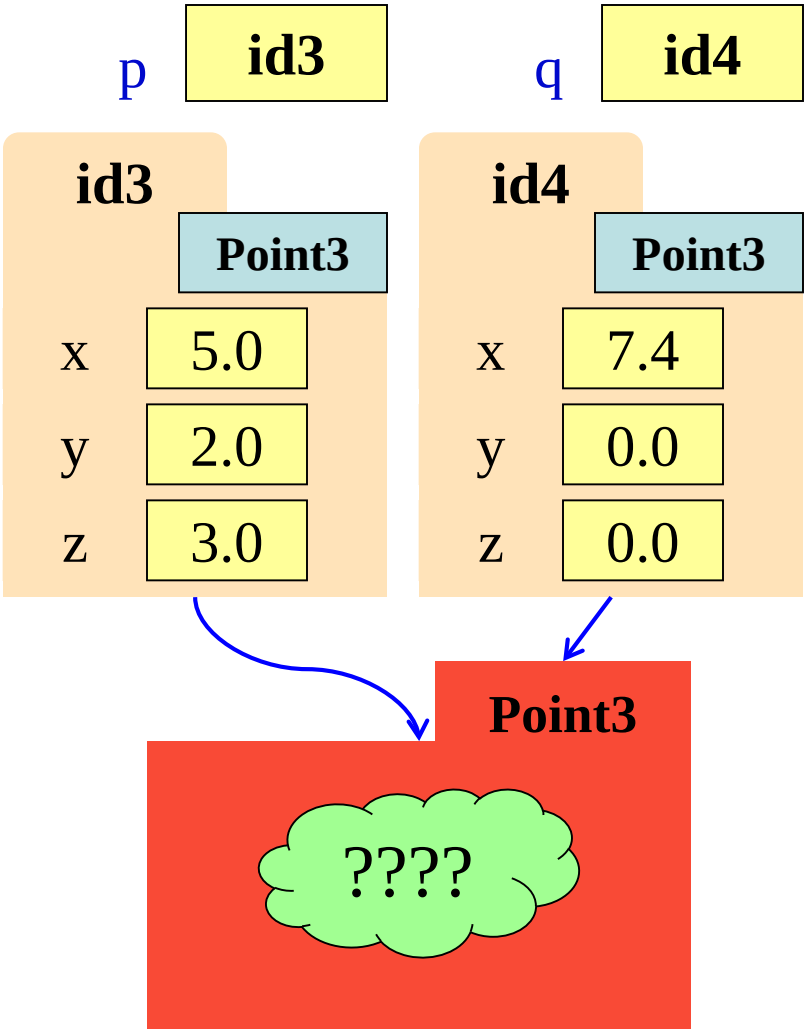
Class Folders

- Data common to all instances



Name Resolution for Objects

- ⟨*object*⟩.⟨*name*⟩ means
  - Go the folder for *object*
  - Find attribute/method *name*
  - If missing, check **class folder**
  - If not in either, raise error
- What is in the class folder?
  - Data common to **all** objects
  - First must understand the ***class definition***



The Class Definition

Goes inside a module, just like a function definition.

class *<class-name>*(object):

```
    """Class specification"""
```

<function definitions>

<assignment statements>

<any other statements also allowed>

Example

```
class Example(object):
    """The simplest possible class."""
    pass
```

The Class Definition

Goes inside a module, just like a function definition.

keyword class
Beginning of a class definition

class *<class-name>*(object):

Just do this.

Specification (similar to one for a function)

"""Class specification"""

<function definitions>

to define **methods**

<assignment statements>

to define **class variables**

<any other statements also allowed>

```
class Example(object):
    """The simplest possible class."""
    pass
```

Example

Python creates after reading the class definition

Important!

YES

```
class Point(object):
    """Instances are 3D points
    x [float]: x coord
    y [float]: y coord
    z [float]: z coord"""
    …
```

3.0-Style Classes
Well-designed

NO

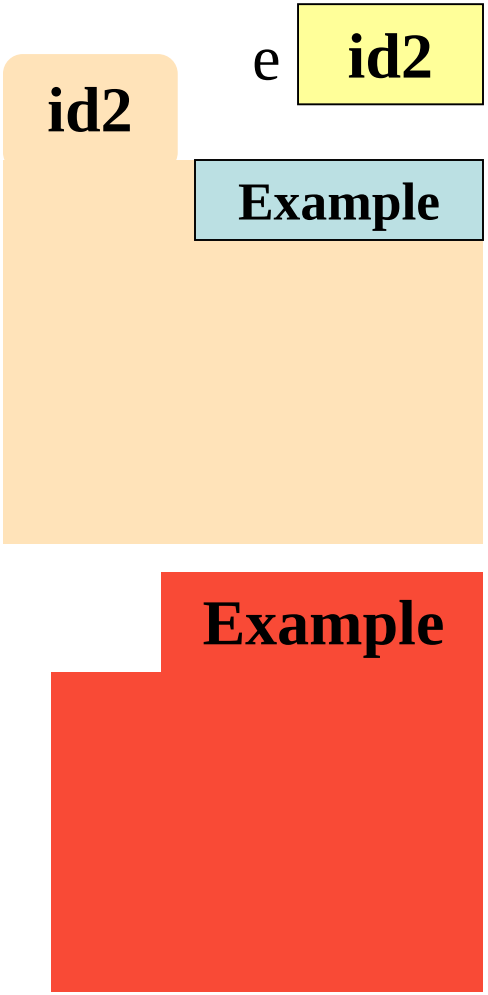
```
class Point:
    """Instances are 3D points
    x [float]: x coord
    y [float]: y coord
    z [float]: z coord"""
    …
```

"Classic" Classes
No reason to use these

Recall: Constructors

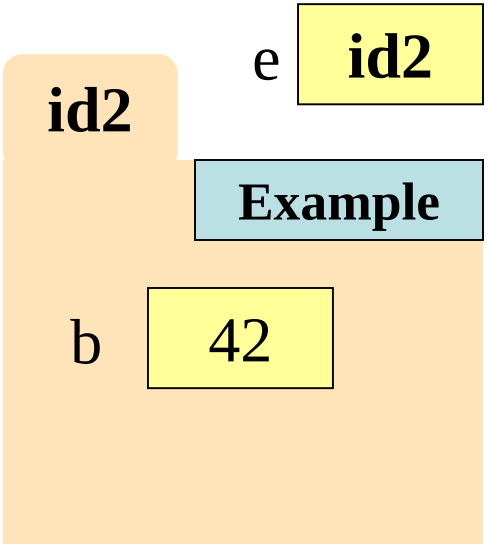
- Function to create new instances
  - Function name == class name
  - Created for you automatically
- Calling the constructor:
  - Makes a new object folder
  - Initializes attributes
  - Returns the id of the folder
- By default, takes no arguments
  - e = Example()

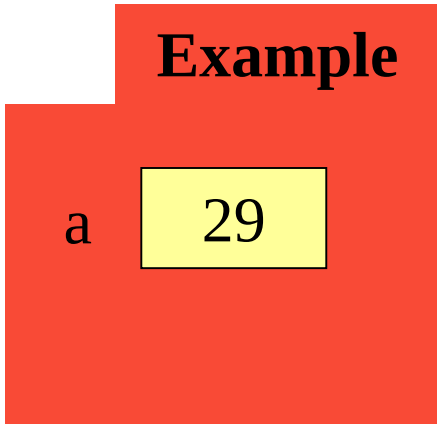
Will come back to this



Instances and Attributes

- Can add object attributes
  - <object>.<att> = <expression>
  - **Example**: e.b = 42
- Can also add class variables
  - <class>.<att> = <expression>
  - **Example**: Example.a = 29
- Can access class attributes through object
  - **Example**: print e.a
  - But assigning it creates object attribute
  - **Example**: e.a = 10
- **Rule**: check object first, then class



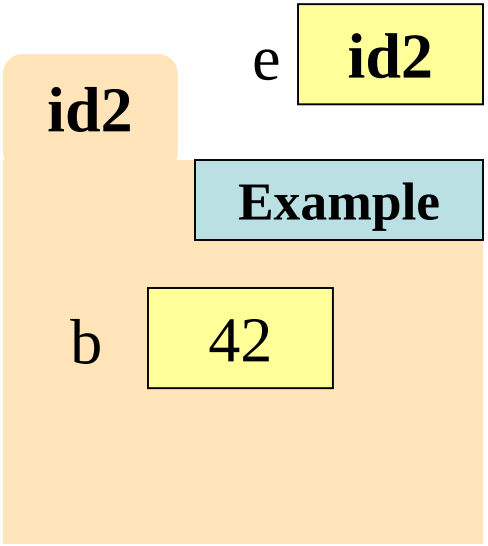


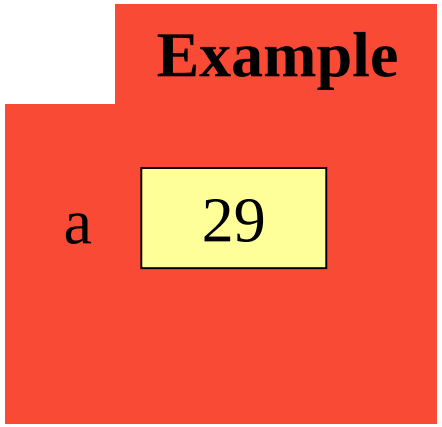
Instances and Attributes

- Can add object attributes
  - <object>.<att> = <expression>
  - **Example**: e.b = 42

Not how usually done

- Can also add class variables
  - <class>.<att> = <expression>
  - **Example**: Example.a = 29
- Can access class attributes through object
  - **Example**: print e.a
  - But assigning it creates object attribute
  - **Example**: e.a = 10
- **Rule**: check object first, then class





What gets Printed?

```
import flower

f = flower.Flower()
g = flower.Flower()

f.robustness = 3
flower.Flower.robustness = 4
flower.Flower.utility = 8
g.utility = 9

print f.robustness
print g.robustness
print f.utility
print g.utility
```

A:
3
4
8
8

B:
4
4
8
9

C:
3
4
8
9

D:
4
4
8
8

Invariants

- Properties of an attribute that must be true
- Works like a precondition:
  - If invariant satisfied, object works properly
  - If not satisfied, object is “corrupted”
- **Examples**:
  - Point3 class: all attributes must be floats
  - RGB class: all attributes must be ints in 0..255
- Purpose of the **class specification**

The Class Specification

```
class Worker(object):
    """An instance is a worker in an organization.

    Instance has basic worker info, but no salary information.

    ATTRIBUTES:
      lname:  Worker’s last name.  [str]
      ssn:    Social security no.   [int in 0..999999999]
      boss:   Worker's boss.        [Worker, or None if no boss]
```

The Class Specification

```
class Worker(object):
    """An instance is a worker in an organization.

    stance has basic worker info, but no salary information.

    ATTRIBUTES:
      lname:  Worker's last name.  [str]
      ssn:    Social security no.  [int in 0..999999999]
      boss:   Worker's boss.       [Worker, or None if no boss]
```

Short summary

More detail

Attribute list

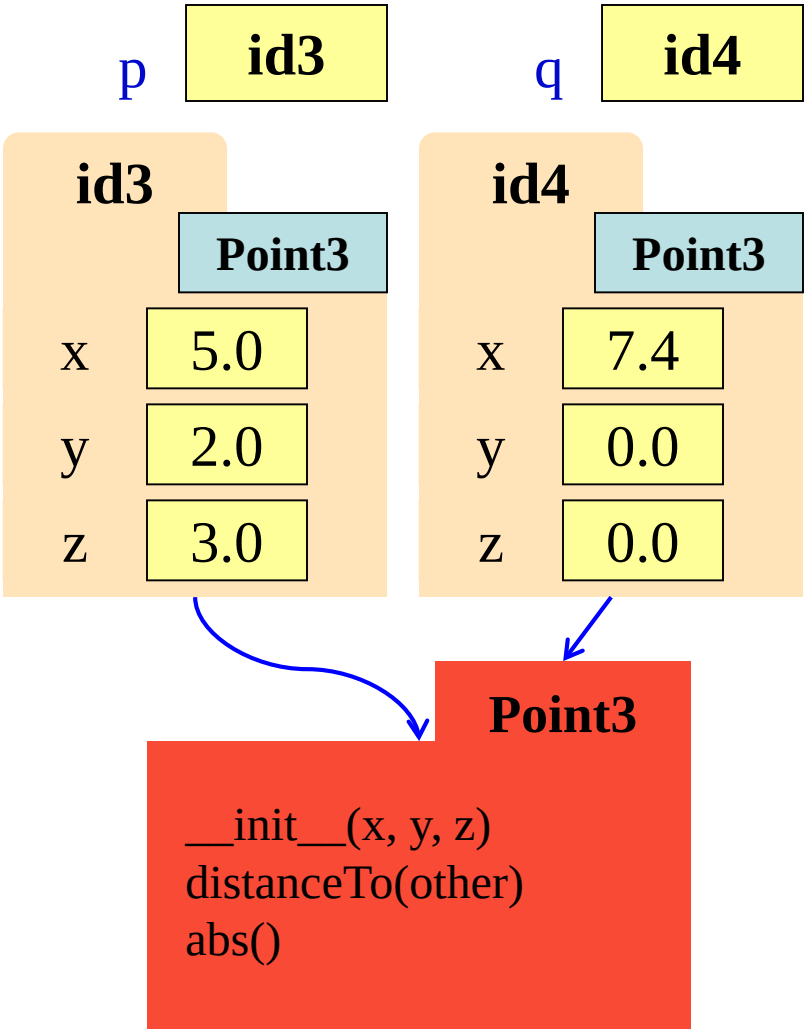
Description

Invariant

Attribute Name

Recall: Objects can have Methods

- **Method**: function tied to object
  - Function call: <function-name>(<arguments>)
  - Method call: <object-variable>.<function-call>
- **Example**: p.distanceTo(q)
- For most Python objects
  - **Attributes** are in **object** folder
  - **Methods** are in **class** folder



Function Definition

- Goal: implement p.distanceTo(q)

Could try to make a function like we have been:

```
def distanceTo(q):
```

Problem: no way to access p

Method Definitions

- Looks like a function def
  - But indented *inside* class
  - The first parameter is always called self
- In a method call:
  - Parentheses have one fewer argument than parameters
  - The object in front is passed to parameter self
- **Example**: a.distanceTo(b)

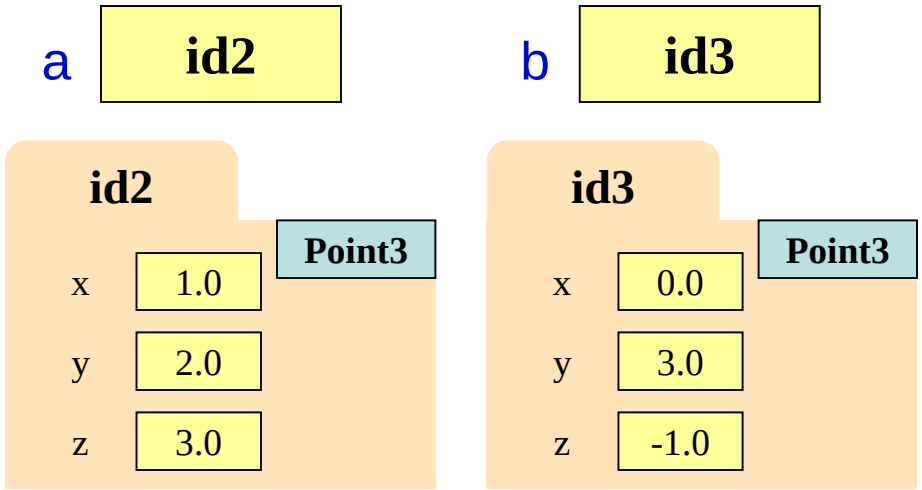
self

q

```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]     """
    def distanceTo(self,q):
        """Returns: dist from self to q
        Precondition: q a Point3"""
        assert type(q) == Point3
        sqrdst = ((self.x-q.x)**2 +
                  (self.y-q.y)**2 +
                  (self.z-q.z)**2)
        return math.sqrt(sqrdst)
```

Method Calls

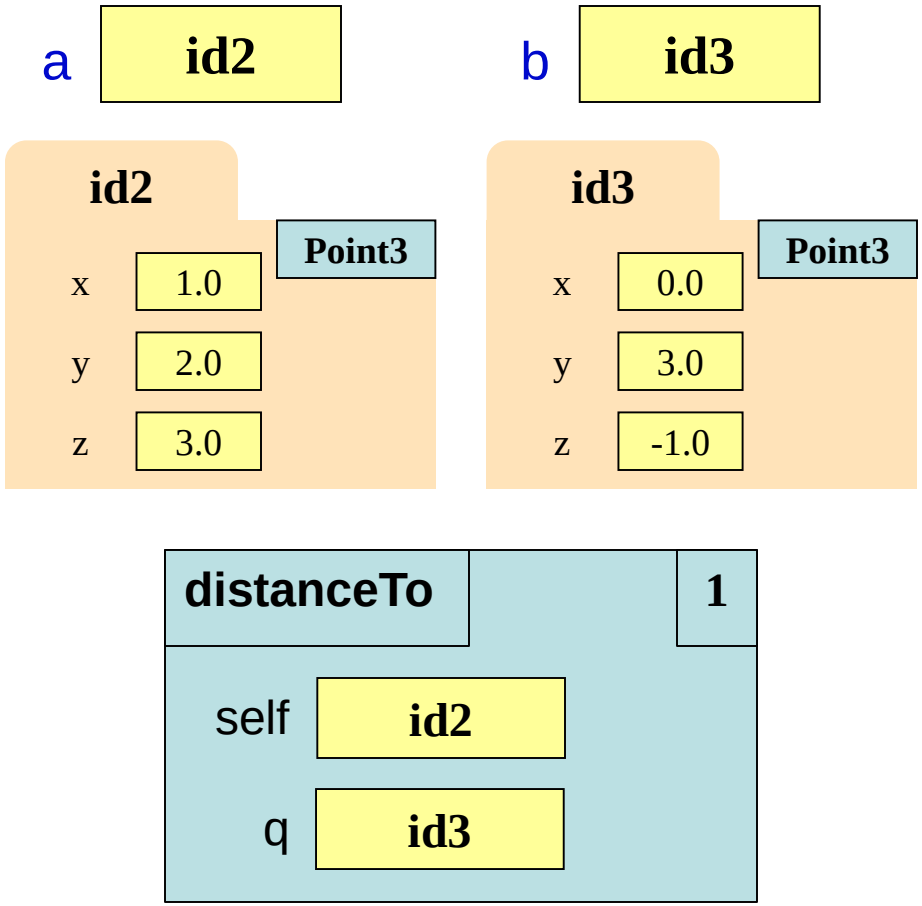
- **Example**: a.distanceTo(b)




```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]    """
    def distanceTo(self,q):
        """Returns: dist from self to q
        Precondition: q a Point3"""
        assert type(q) == Point3
        sqrdst = ((self.x-q.x)**2 +
                  (self.y-q.y)**2 +
                  (self.z-q.z)**2)
        return math.sqrt(sqrdst)
```

Method Calls

- **Example**: a.distanceTo(b)




```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]    """
    def distanceTo(self,q):
        """Returns: dist from self to q
        Precondition: q a Point3"""
        assert type(q) == Point3
        sqrdst = ((self.x-q.x)**2 +
                  (self.y-q.y)**2 +
                  (self.z-q.z)**2)
        return math.sqrt(sqrdst)
```

Don’t forget self!

```
p = Point3(1.0, 2.0, 3.0)
q = Point3(4.0, 5.0, 6.0)
print p.distanceTo(q)
```

```
def distanceTo(other):
    sqrdst = ((x-other.x)**2 +
        (y-other.y)**2 +
        (z-other.z)**2)
    return math.sqrt(sqrdst)
```

????

TypeError: distanceTo() takes exactly 1 argument (2 given)

<var>.<method_name> **always** passes <var> as first argument

Don’t forget self!

```
p = Point3(1.0, 2.0, 3.0)
q = Point3(4.0, 5.0, 6.0)
print p.distanceTo(q)
```

```
def distanceTo(self, other):
    sqrdst = ((x-other.x)**2 +
        (y-other.y)**2 +
        (z-other.z)**2)
    return math.sqrt(sqrdst)
```

Methods can’t access object attributes without self.

NameError: global name 'x' is not defined

Initializing the Attributes of an Object (Folder)

- Creating a new Worker is a multi-step process:
  - w = Worker() ← Instance is empty
  - w.lname = 'Andersen'
  - …
- Want to use something like

  w = Worker('Andersen', 1234, None)

  - Create a new Worker **and** assign attributes
  - lname to 'Andersen', ssn to 1234, and boss to None
- Need a **custom constructor**

Special Method: __init__

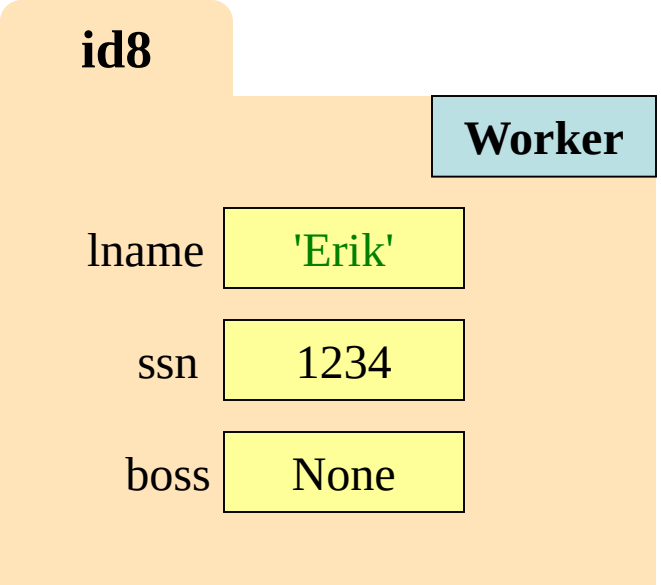
```
w = Worker('Andersen', 1234, None)
```

Called by the constructor

```
def __init__(self, n, s, b):
    """Initializer: creates a Worker

    Has last name n, SSN s, and boss b

    Precondition: n a string, s an int in
    range 0..999999999, and b either
    a Worker or None.
    self.lname = n
    self.ssn  = s
    self.boss = b
```



Special Method: __init__

```
w = Worker('Anders', 1234, None)
```

two underscores

don't forget self

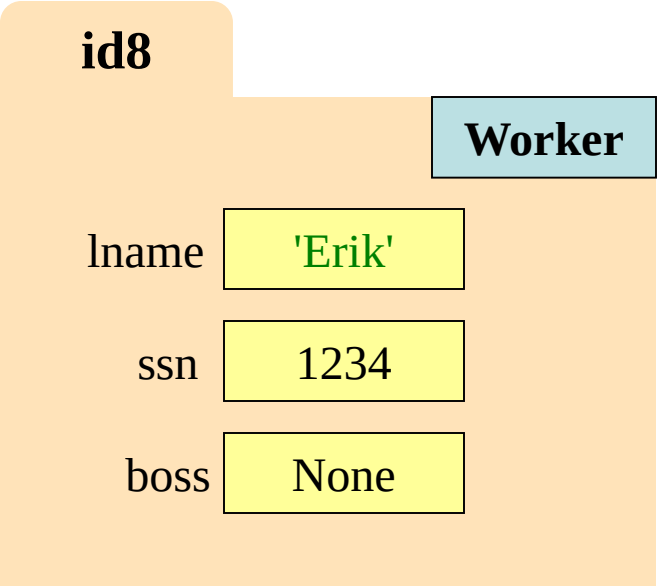
```
def __init__(self, n, s, b):
    """Initializer: creates a Worker

    Has last name n, SSN s, and boss b

    Precondition: n a string, s an int in
    range 0..999999999, and b either
    a Worker or None.
    self.lname = n
    self.ssn  = s
    self.boss = b
```

use self to assign attributes

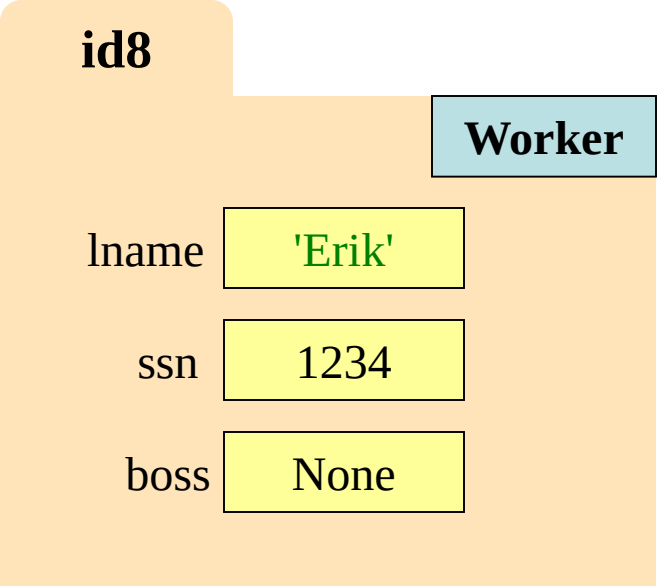
Called by the constructor



Evaluating a Constructor Expression

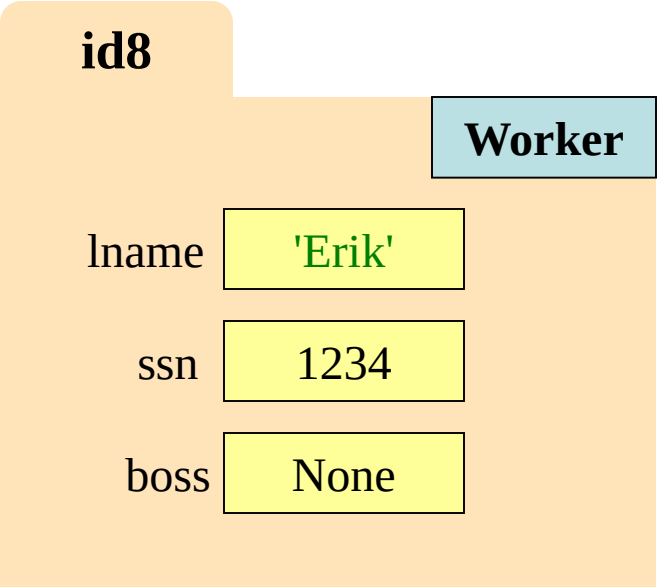
Worker('Erik', 1234, None)

1. Creates a new object (folder) of the class Worker
   - Instance is initially empty
2. Puts the folder into heap space
3. Executes the method __init__
   - Passes folder name to self
   - Passes other arguments in order
   - Executes the (assignment) commands in initializer body
4. Returns the object (folder) name



Aside: The Value None

- The boss field is a problem.
  - boss refers to a Worker object
  - Some workers have no boss
  - Or maybe not assigned yet
- **Solution**: use value None
  - **None**: Lack of (folder) name
  - Will reassign the field later!



Making Arguments Optional

- We can assign default values to __init__ arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional
- **Examples**:
  - p = Point3() # (0,0,0)
  - p = Point3(1,2,3) # (1,2,3)
  - p = Point3(1,2) # (1,2,0)
  - p = Point3(y=3) # (0,3,0)
  - p = Point3(1,z=2) # (1,0,2)

```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]    """

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes a new Point
        Precondition: x,y,z are numbers"""
        self.x = x
        self.y = y
        self.z = z
…
```

Making Arguments Optional

- We can assign default values to __init__ arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional
- **Examples**:
  - p = Point3() # (0,0,0)
  - p = Point3(... 3)
  - p = Point3(1,2)
  - p = Point3(y=3) # (0,3,0)
  - p = Point3(1,z=2)

Assigns in order

Use parameter name when out of order

Can mix two approaches

```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]    """

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes a new Point
        Precondition: x,y,z are numbers"""
        self.x = x
        self.y = y
        self.z = z
```

Making Arguments Optional

- We can assign default values to __init__ arguments
  - Write as assignments to parameters in definition
  - Parameters with default values are optional
- **Examples**:
  - p = Point3() # (0,0,0)
  - p = Point3(3)
  - p = Point3(1,2)
  - p = Point3(y=3) # (0,3,0)
  - p = Point3(1,z=2)

Assigns in order

Use parameter name when out of order

Can mix two approaches

```
class Point3(object):
    """Instances are points in 3d space
        x: x coord [float]
        y: y coord [float]
        z: z coord [float]    """

    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes a new Point
        Precondition: x,y,z are numbers"""
        self.x = x
        self.y = y
```

Not limited to methods. Can do with any function.