Lecture 11

Defining Classes

Announcements

Reading

- Chapters 15, 16
  - Chapter 17 for Friday
  - Critical for this topic

Assignment 2

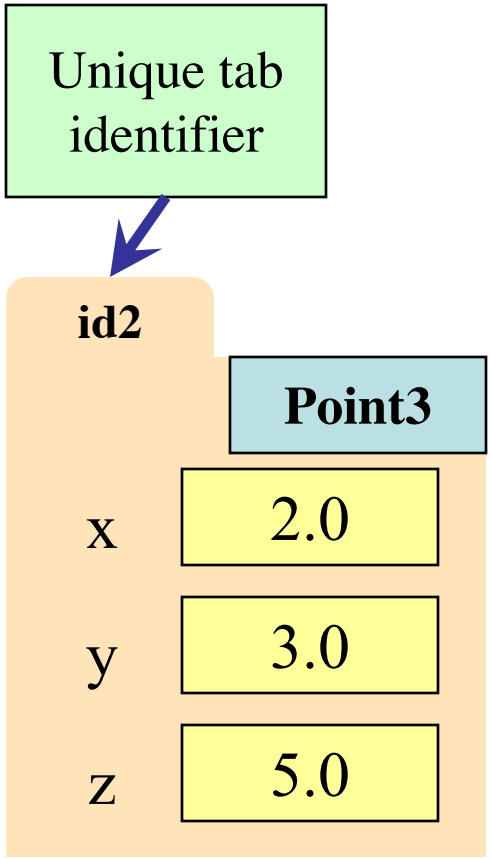
- Due Thursday, Oct 13
  - Need 85% to pass
  - Otherwise, you will revise

Labs

- Last lab is this week
  - Due by next Wednesday
  - Can check off next week
- **Recall**: Can miss one lab
  - Can skip this if did others
  - Else, must do this one
- Part of lab useful for A2
  - First practice with objects

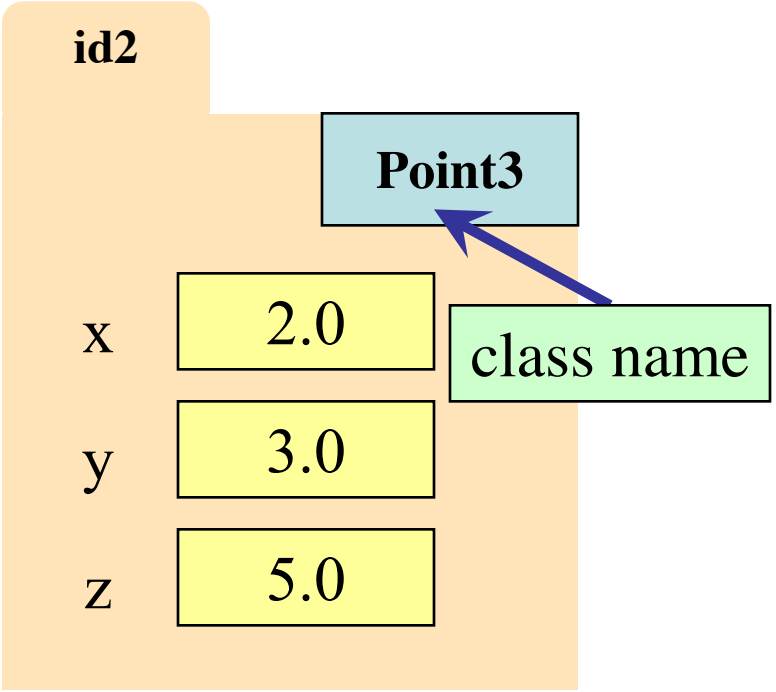
Recall: Objects as Data in Folders

- 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

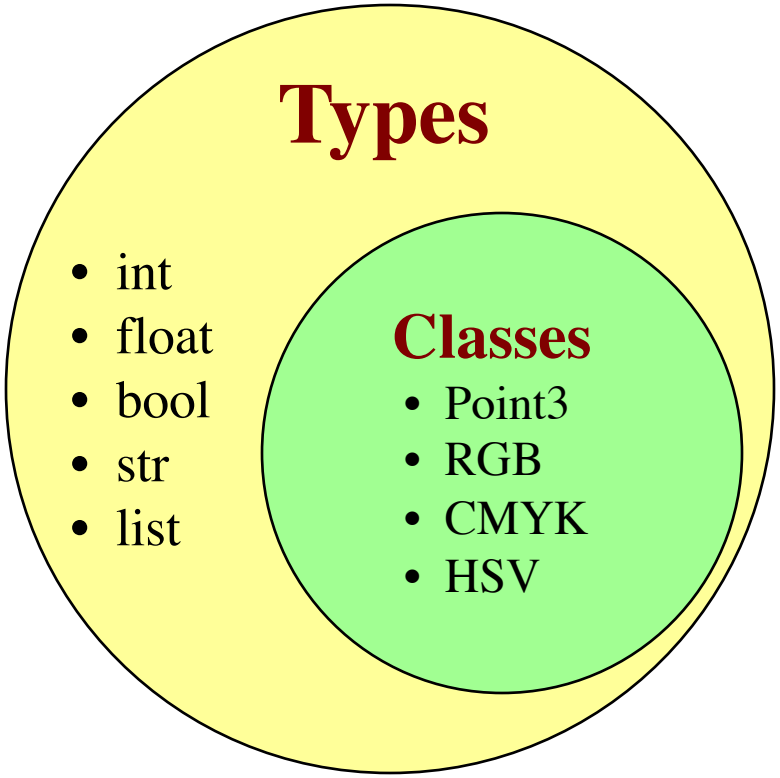


Recall: Classes are Types for Objects

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



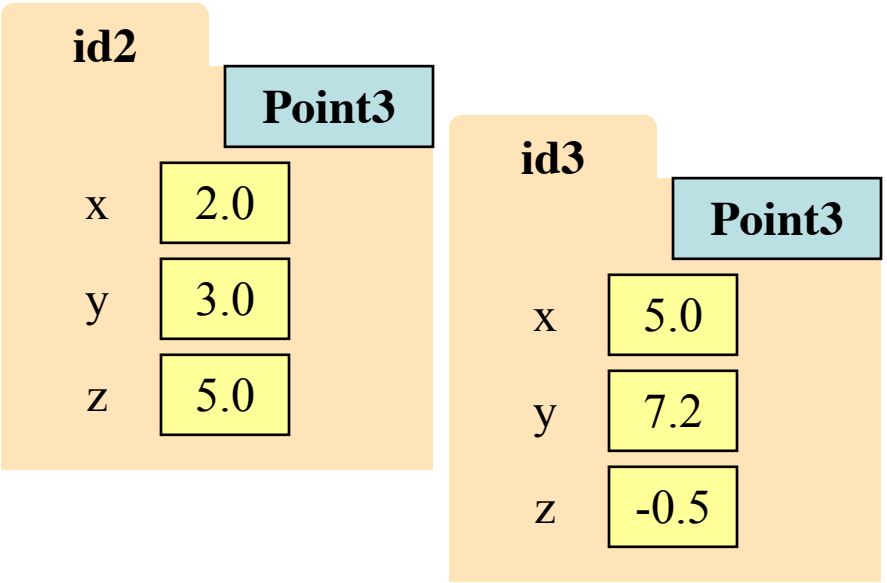
- Classes are how we add new types to Python



Classes Have Folders Too

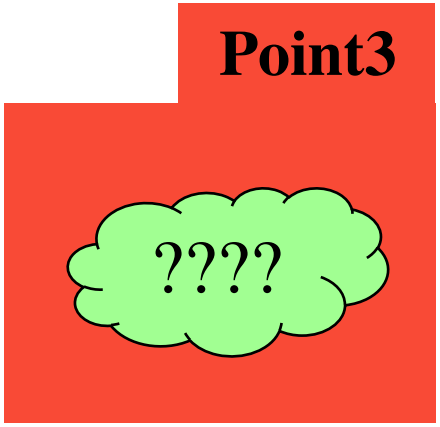
Object Folders

- Separate for each *instance*



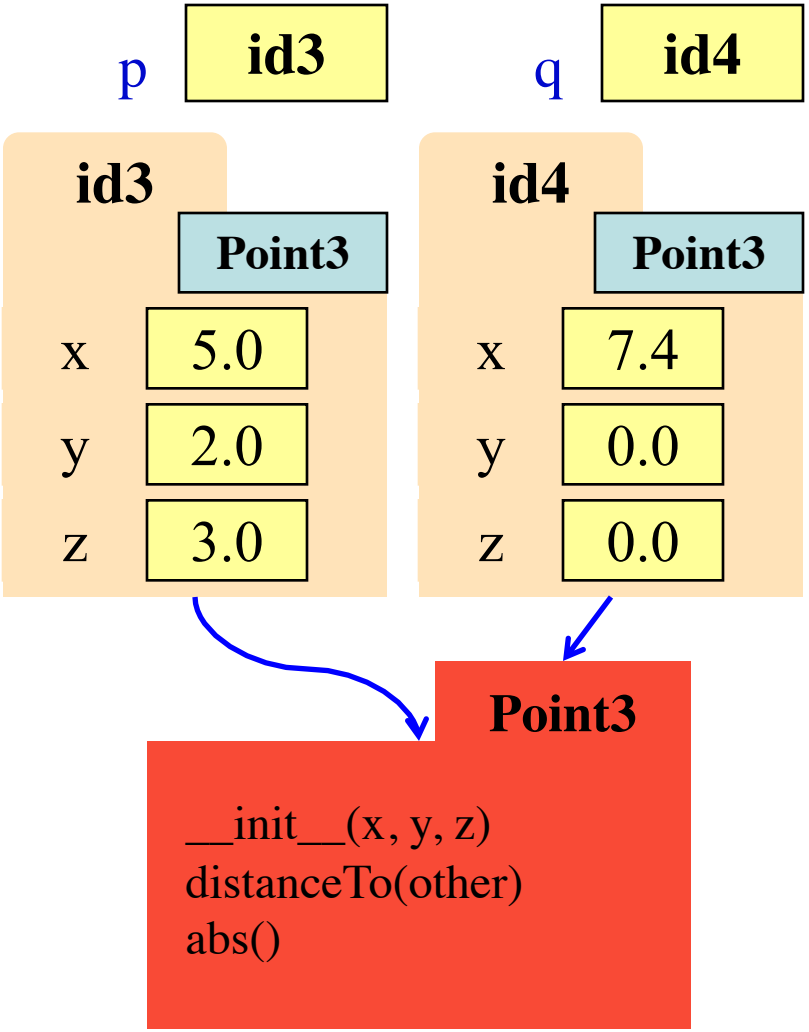
Class Folders

- Data common to all instances



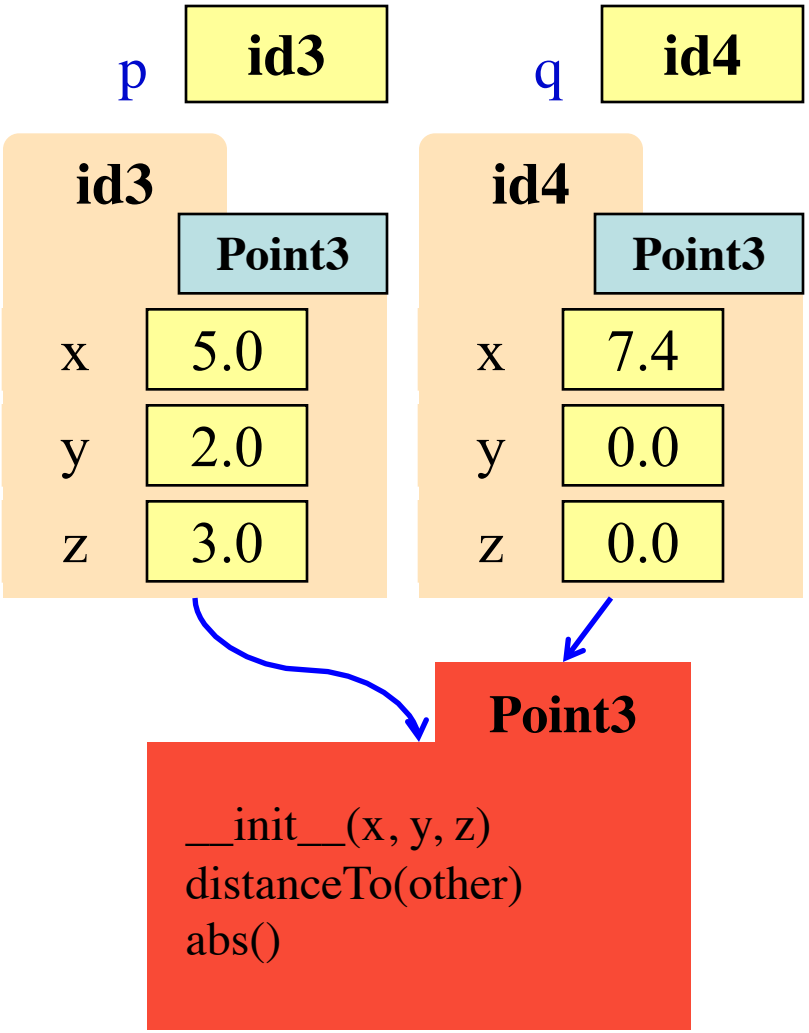
Recall: Objects can have Methods

- **Method**: function tied to object
  - Function call: <function-name>(<arguments>)
  - Method call: <object-variable>.<function-call>
  - Use of a method is a *method call*
- **Example**: `p.distanceTo(q)`
  - Both `p` and `q` act as arguments
  - Very much like `distanceTo(p, q)`
- Methods (often) in **class folders**



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
- For most Python objects
  - **Attributes** are in **object** folder
  - **Methods** are in **class** folder
- But rules can be broken…



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):

Do not forget the colon!

more on this later

Specification (similar to one for a function)

"""Class specification"""

<*function definitions*>

to define **methods**

<*assignment statements*>

…but not often used

to define **variables**

<*any other statements also allowed*>

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

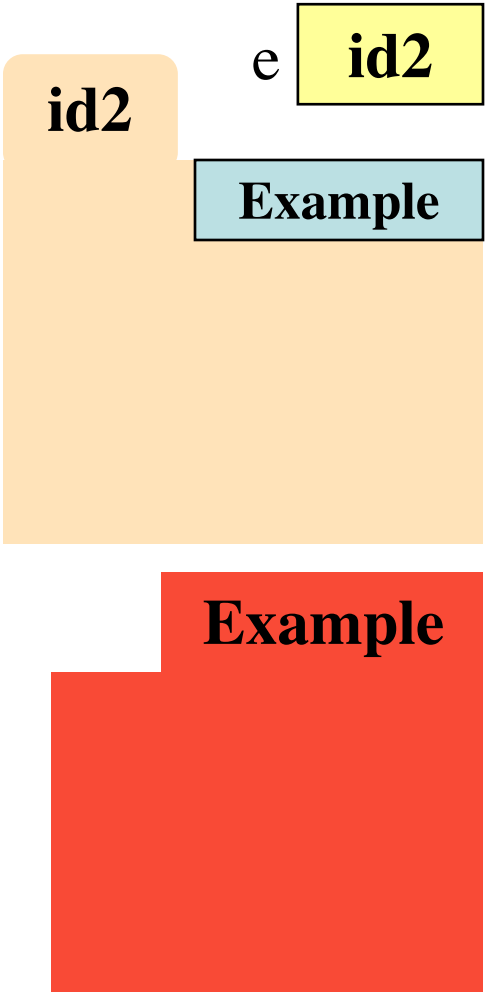
Example

Python creates after reading the class definition

Recall: Constructors

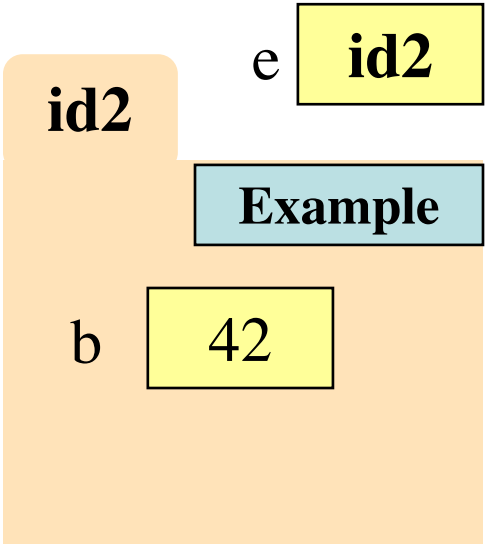
- Function to create new instances
  - Function name == class name
  - Created for you automatically
- Calling the constructor:
  - Makes a new object folder
  - Initializes attributes
  - Returns the id of the folder
- By default, takes no arguments
  - `e = Example()`

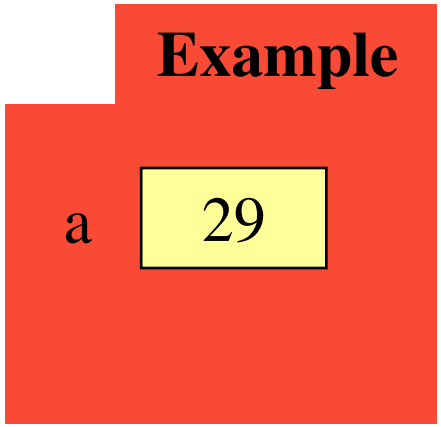
Will come back to this



Instances and Attributes

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



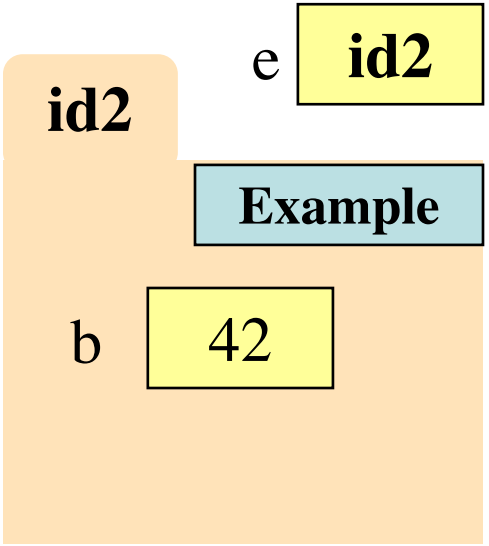


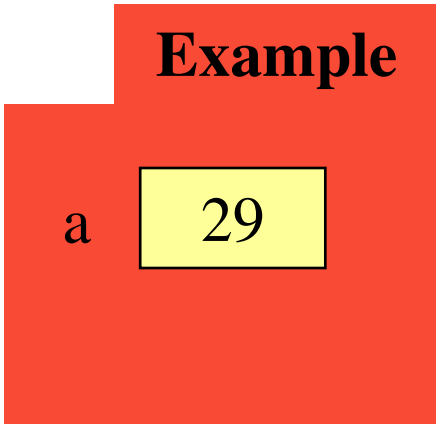
Instances and Attributes

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

Not how usually done

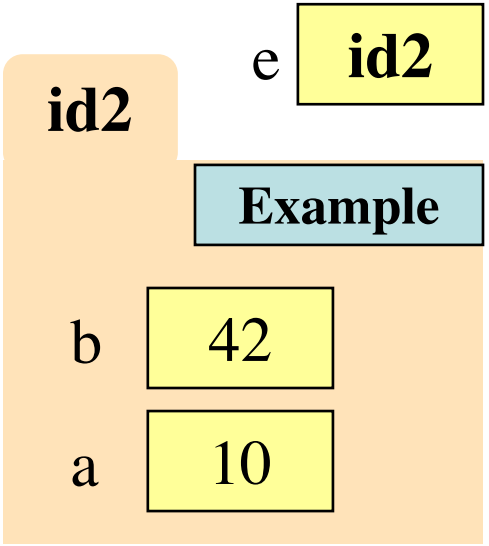
- Assignments can add class attributes
  - <class>.<att> = <expression>
  - **Example**: `Example.a = 29`
- Objects can access class attributes
  - **Example**: `print e.a`
  - But assigning it creates object attribute
  - **Example**: `e.a = 10`
- **Rule**: check object first, then class

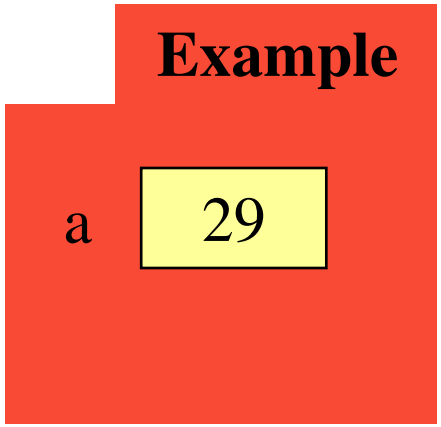




Instances and Attributes

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





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**:
  - Point 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.

    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]
```

Short summary

More detail

Attribute list

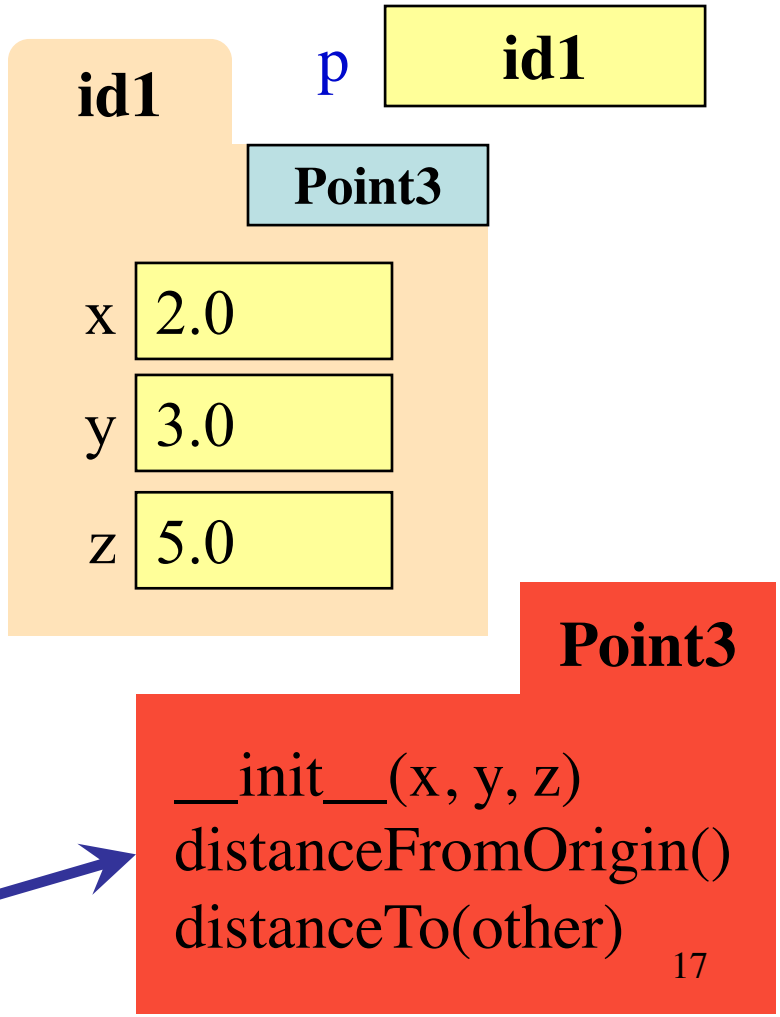
Description

Invariant

Attribute Name

Recall: Objects can have Methods

- **Method**: function tied to object
  - Function call: <function-name>(<arguments>)
  - Method call: <object-variable>.<function-call>
  - Use of a method is a *method call*
- **Example**: `p.distanceTo(q)`
  - Both `p` and `q` act as arguments
  - Very much like `distanceTo(p, q)`



Methods are organized with the class

Methods

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

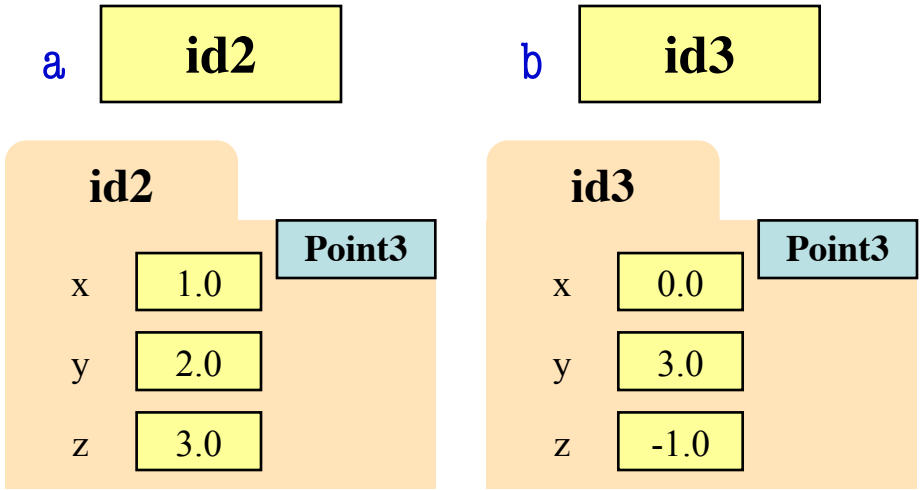
self

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)
```

Methods Calls

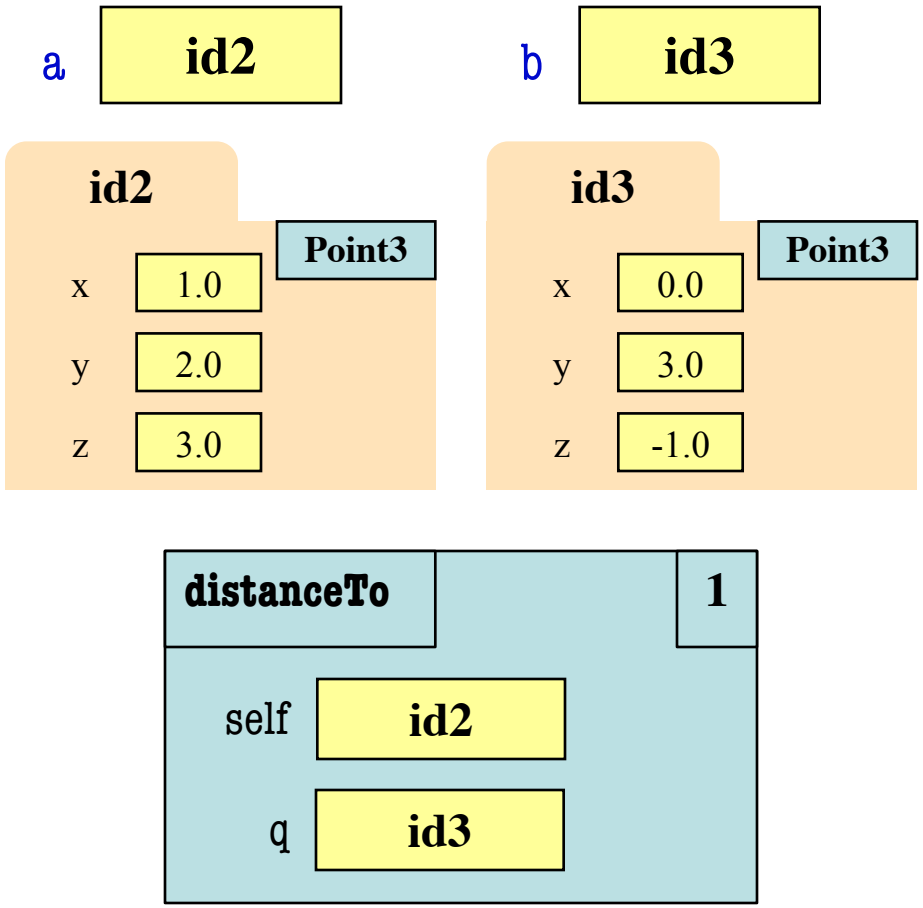
- **Example**: a.distanceTo(b)




```
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)
```

Methods 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)
```

Initializing the Attributes of an Object (Folder)

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

  `w = Worker('White', 1234, None)`
  - Create a new Worker **and** assign fields
  - `lname` to 'White', `ssn` to 1234, and `boss` to None
- Need a **custom constructor**

Special Method: __init__

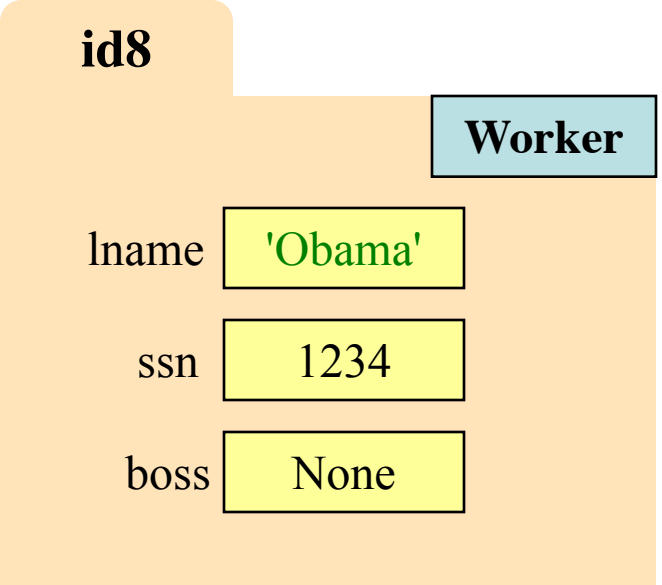
```
w = Worker('Obama', 1234, None)
```

```
def __init__(self, n, s, b):
    """Constructor: 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
```

Called by the constructor



Special Method: __init__

two underscores

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

don't forget self

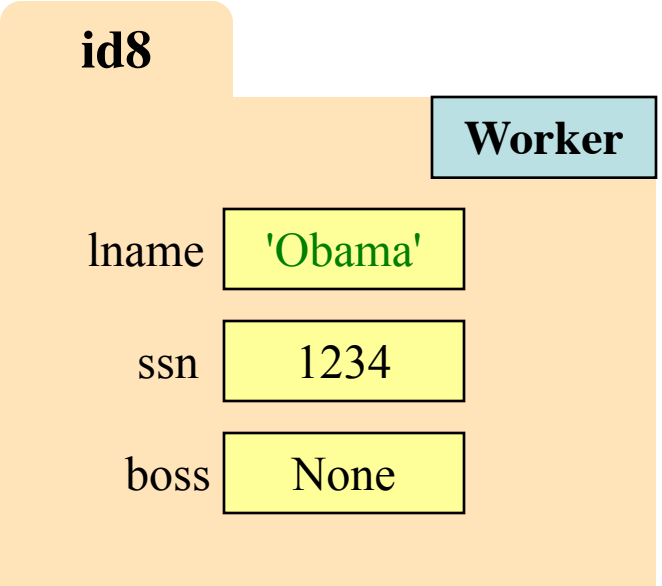
```
def __init__(self, n, s, b):
    """Constructor: creates a Worker

    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 access fields

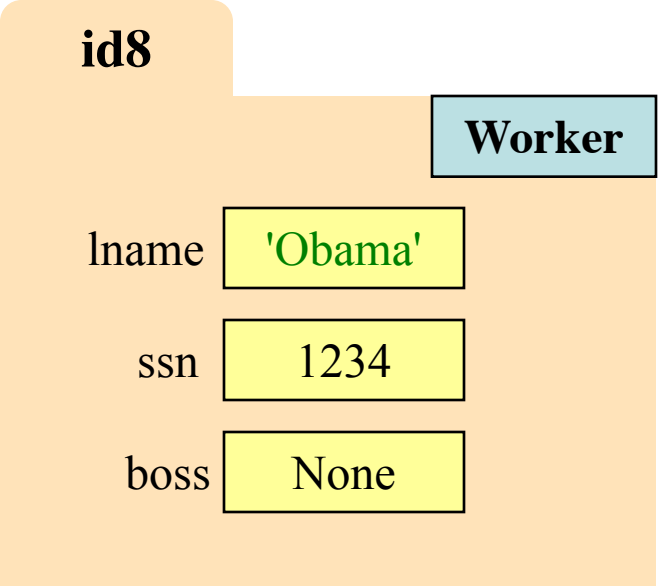
Called by the constructor



Evaluating a Constructor Expression

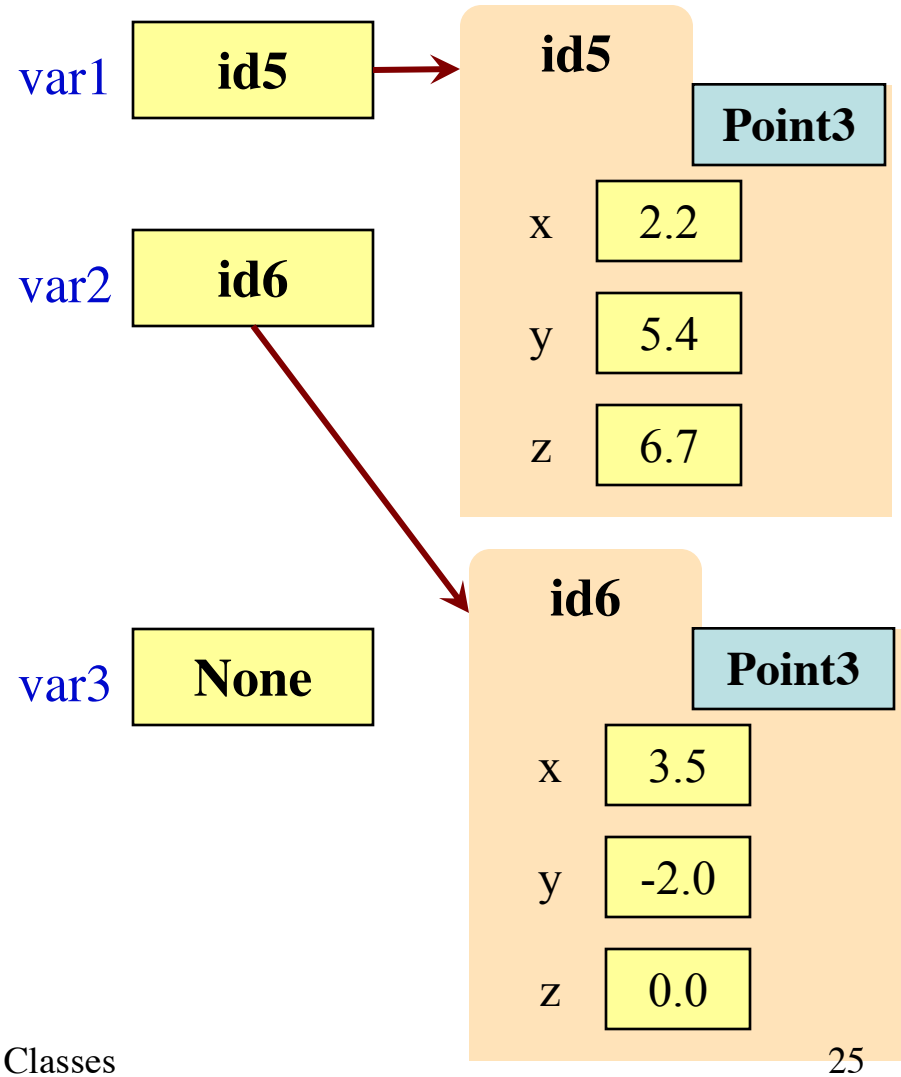
```
Worker('Obama', 1234, None)
```

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 constructor 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 (the buck stops there)
- **Solution**: use value `None`
  - **`None`**: Lack of (folder) name
  - Will reassign the field later!
- Be careful with `None` values
  - var3.x gives error!
  - There is no name in var3
  - Which Point3 to use?



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 = 0.0 # x coord, float
    y = 0.0 # y coord, float
    z = 0.0 # z coord, float

    def __init__(self,x=0,y=0,z=0):
        """Constructor: 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(
  - 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):
        """Constructor: 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(
  - 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):
        """Constructor: makes a po
        Precondition
```

Not limited to methods.
Can do with any function.

What Does str() Do On Objects?

- Does **NOT** display contents

```
>>> p = Point3(1,2,3)
>>> str(p)
'<Point object at 0x1007a90>'
```

- To display contents, you must implement a special method
  - __str__ for str()
  - __repr__ for backquotes
  - If only implement __str__, backquotes do not work
  - If implement __repr__ but not __str__, str() uses it too

```
class Point3(object):
    """Instances are points in 3d space"""
    ...
    def __str__(self):
        """Returns: string with contents"""
        return '('+self.x + ',' +
                  self.y + ',' +
                  self.z + ')'

    def __repr__(self):
        """Returns: unambiguous string"""
        return str(self.__class__)+
                str(self)
```

What Does str() Do On Objects?

- Does **NOT** display contents

```
>>> p = Point3(1,2,3)
>>> str(p)
'<Point object at 0x1007a90>'
```

- To display contents, you must implement a special method
  - __str__ for str()
  - __repr__ for backquotes
  - If only implement __str__, backquotes do not work
  - If implement __repr__ but not __str__, str() uses it too

```
class Point3(object):
    """Instances are points in 3d space"""
    ...
    def __str__(self):
        """Returns: string with contents"""
        return '('+self.x + ',' +
                   self.y + ',' +
                   self.z + ')'

    def __repr__(self):
        """Returns: unambiguous string"""
        return str(self.__class__)+
               str(self)
```

Gives the class name

__repr__ using __str__ as helper

Important!

YES

```
class Point3(object):
    """Instances are 3D points
    Attributes:
        x: x-coord [float]
        y: y-coord [float]
        z: z-coord [float]"""
    ...
```

3.0-Style Classes
Well-Designed

NO

```
class Point3:
    """Instances are 3D points
    Attributes:
        x: x-coord [float]
        y: y-coord [float]
        z: z-coord [float]"""
    ...
```

"Old-Style" Classes
Very, Very Bad