

Lecture 16

# Classes

# Announcements for This Lecture

---

## Prelim and Regrades

---

- All Prelims are all done
  - Solutions are now posted
  - Make-ups graded by Sun.
- Handed back in lab section
- Regrades open next week
  - Only for MAJOR mistakes
  - We reserve the right to take off points in a regrade

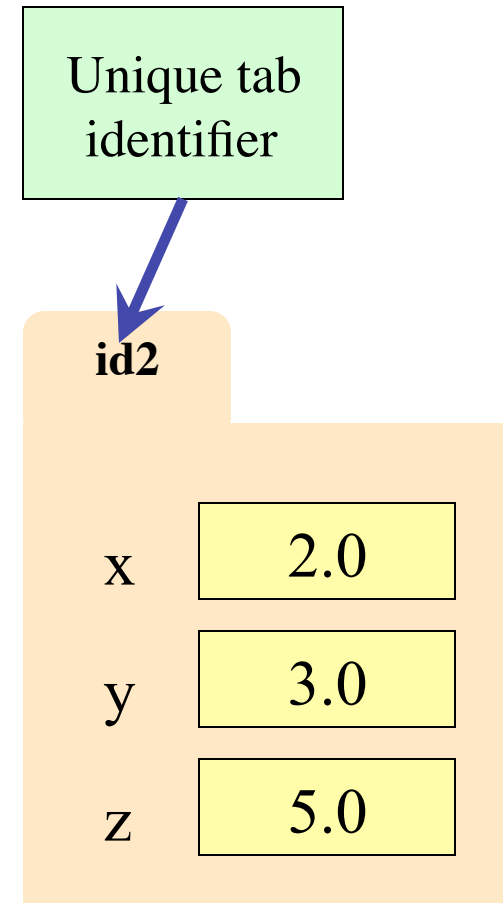
## Assignments/Reading

---

- Should be working on A4
  - Watch Piazza for updates
  - Complete Part B tomorrow
  - Recursion next week
- **Reading:** Chapters 15, 16
  - Chapter 17 for next week
  - Lot of reading but *important*

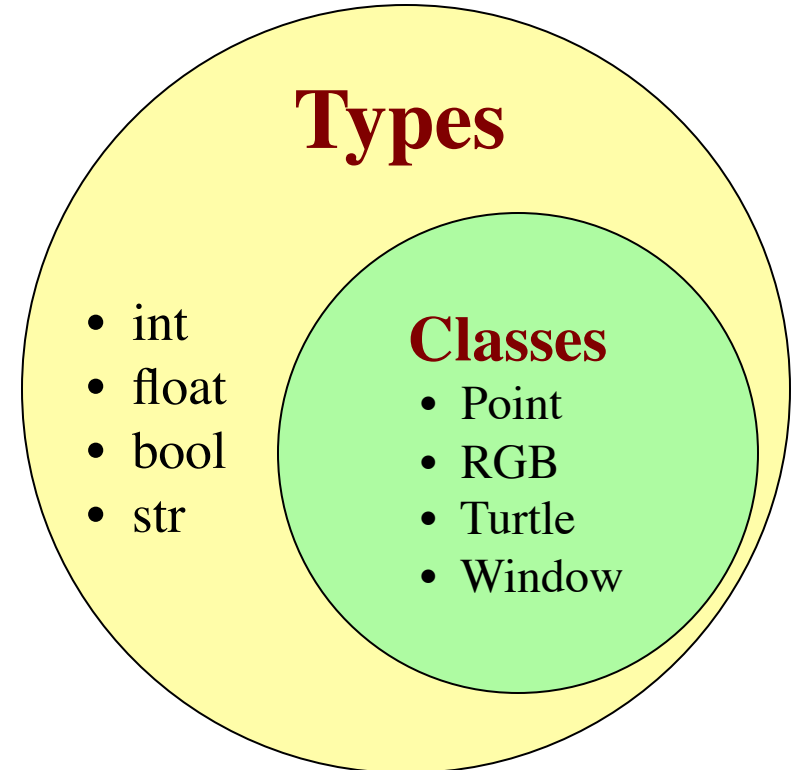
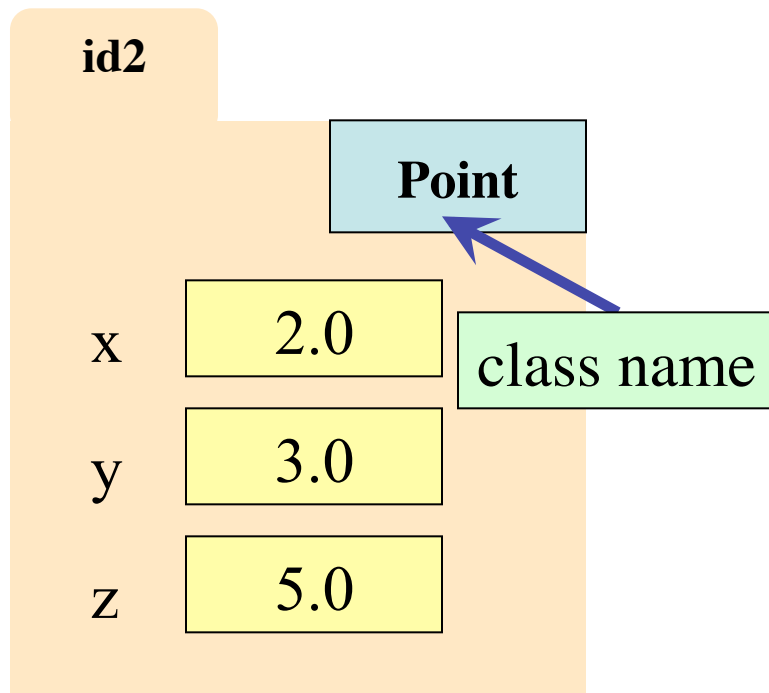
# Recall: Objects as Data in Folders

- 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

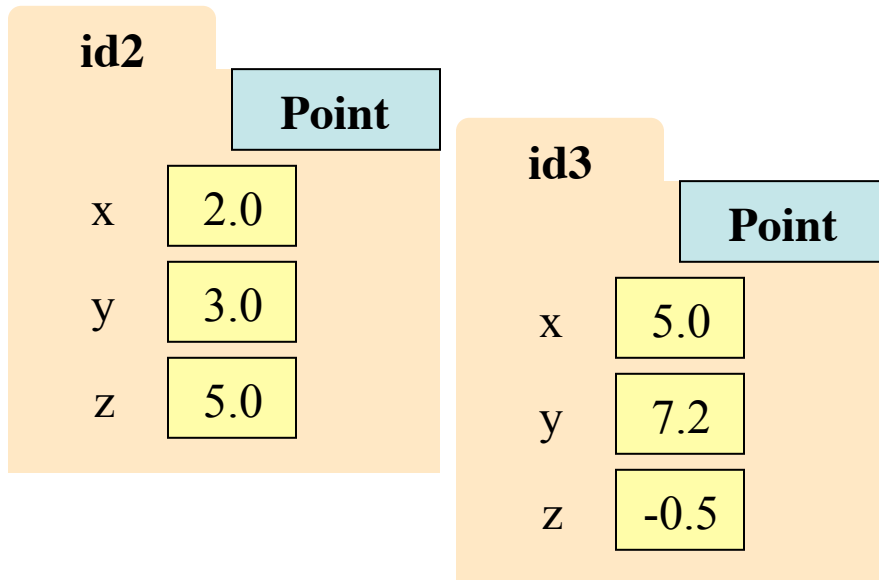
- Values must have a type
  - An object is a **value**
  - Object type is a **class**
- Classes are how we add new types to Python



# Classes Have Folders Too

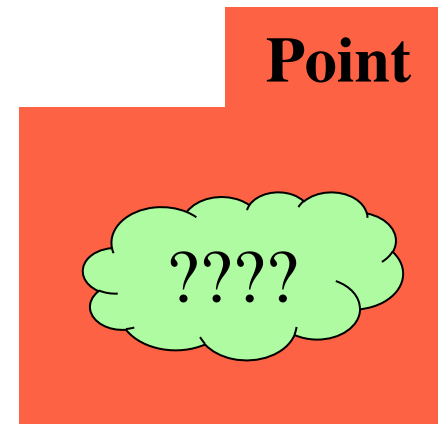
## Object Folders

- Separate for each *instance*



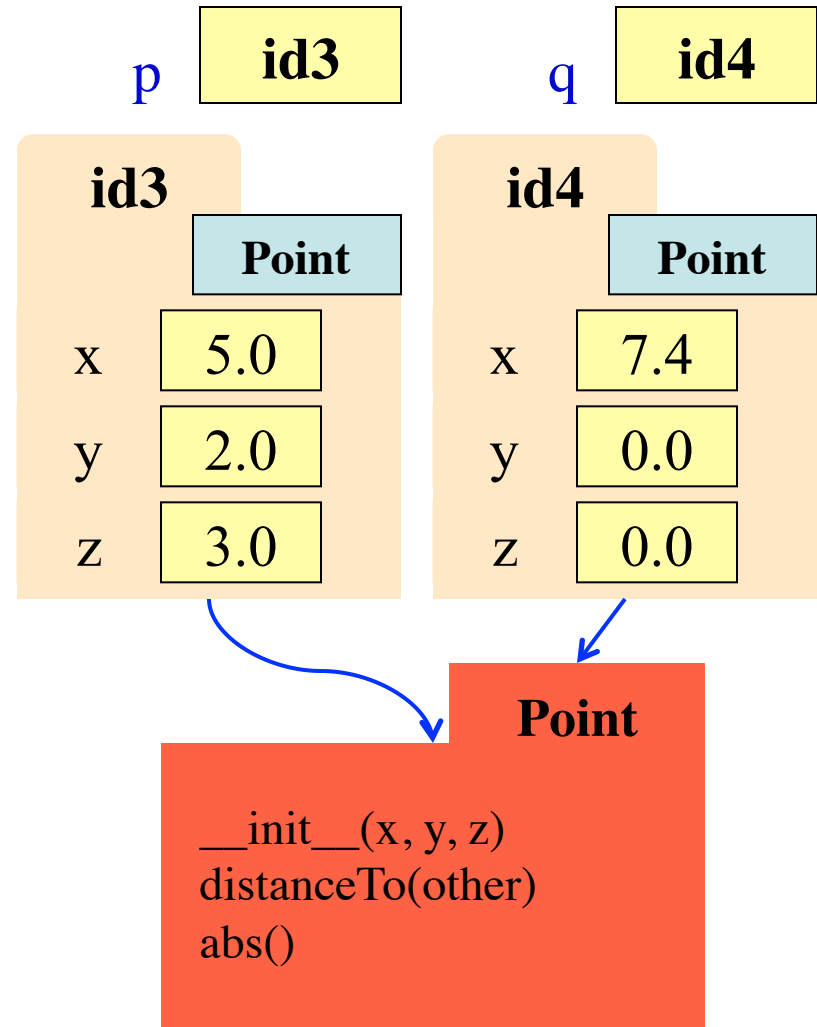
## Class Folders

- Data common to all instances



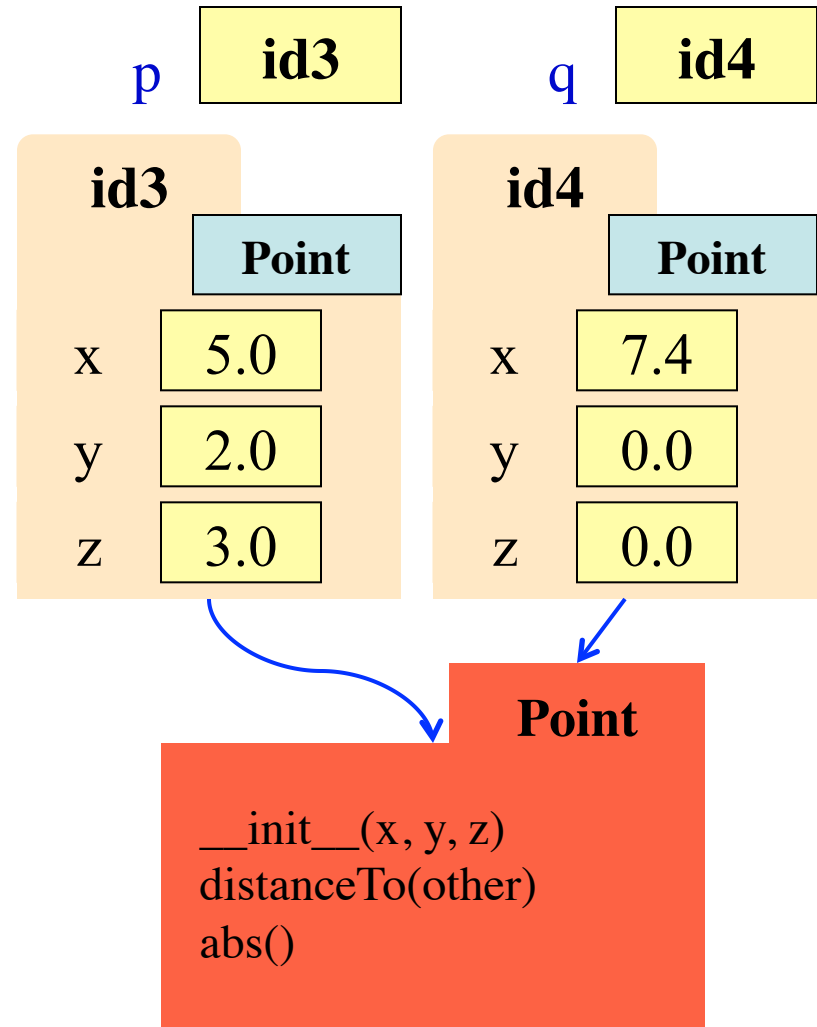
# Recall: Objects can have Methods

- **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

- $\langle object \rangle.\langle name \rangle$  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
- Rules can be broken...  
(but not in this class)



# 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>*

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

**Example**



# 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!

Specification  
(similar to one for a function)

**"""Class specification"""**

more on this later

*<function definitions>*

to define **methods**

*<assignment statements>*

...but not often used

to define **attributes**

*<any other statements also allowed>*

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

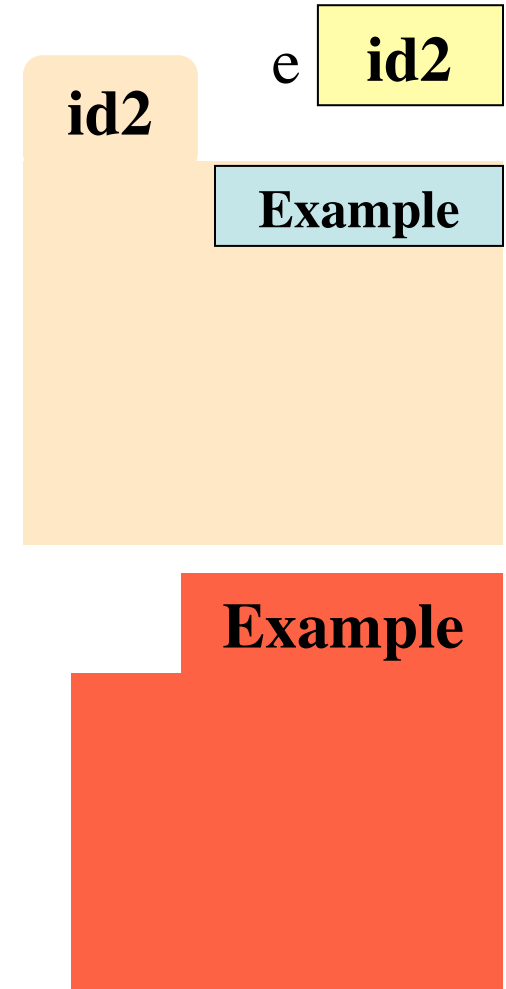
## Example

Python creates after reading the class definition

# Recall: Constructors

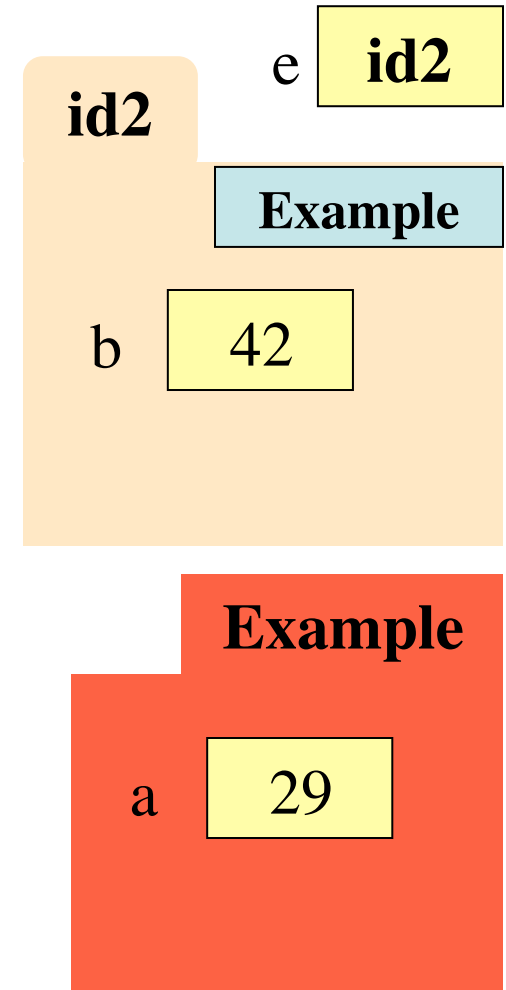
- Function to create new instances
  - Function name == class name
  - Created for you automatically
- Calling the constructor:
  - Makes a new object folder
  - Initializes attributes
  - Returns the id of the folder
- By default, takes no arguments
  - `e = Example()`

Will come back to this



# Instances and Attributes

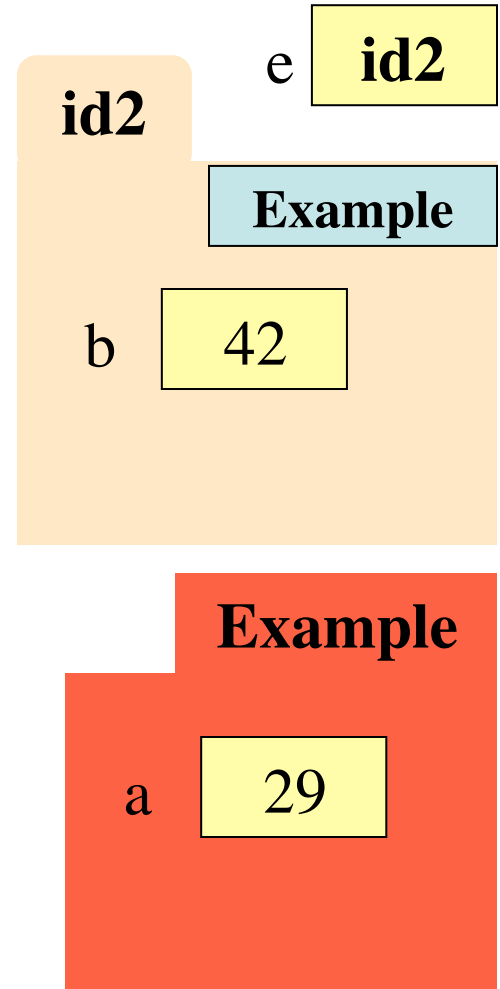
- Assignments add object attributes
  - `<object>.<att> = <expression>`
  - **Example:** `e.b = 42`
- Assignments can add class attributes
  - `<class>.<att> = <expression>`
  - **Example:** `Example.a = 29`
- Objects can access class attributes
  - **Example:** `print e.a`
  - But assigning it creates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



# Instances and Attributes

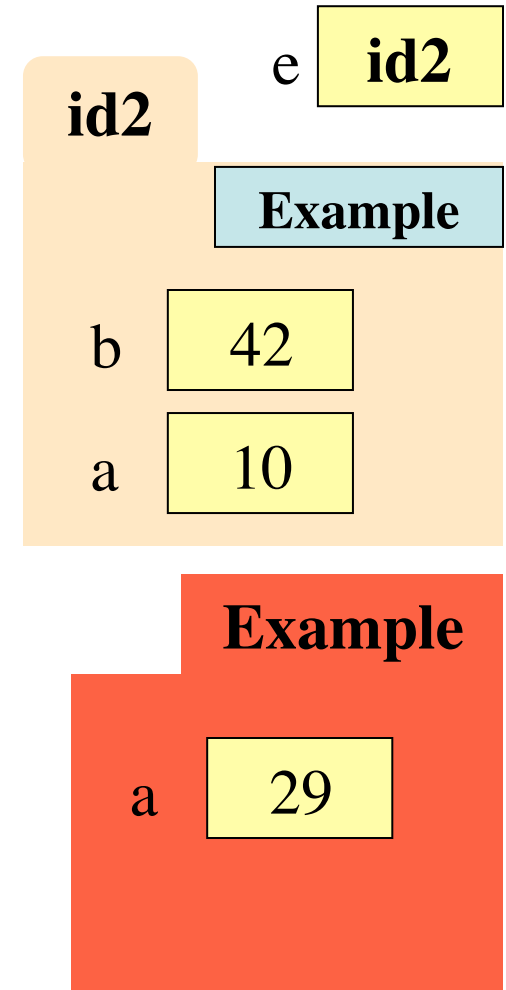
- Assignments add object attributes
  - `<object>.<att> = <expression>`
  - **Example:** `e.b = 42`
- Assignments can add class attributes
  - `<class>.<att> = <expression>`
  - **Example:** `Example.a = 29`
- Objects can access class attributes
  - **Example:** `print e.a`
  - But assigning it creates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

Not how usually done



# 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):
```

Short  
summary

```
    """An instance is a worker in an organization.
```

More  
detail

Attribute  
list

```
    Instance has basic worker info, but no salary information.
```

Description

```
    ATTRIBUTES:
```

Invariant

Attribute  
Name

```
    lname: Worker's last name. [str]
```

```
    ssn: Social security no. [int in 0..999999999]
```

```
    boss: Worker's boss. [Worker, or None if no boss]
```



# 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 less argument than parameters
  - The object in front is passed to parameter self
- **Example:** `a.distanceTo(b)`

self

q

```
class Point(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 Point"""
```

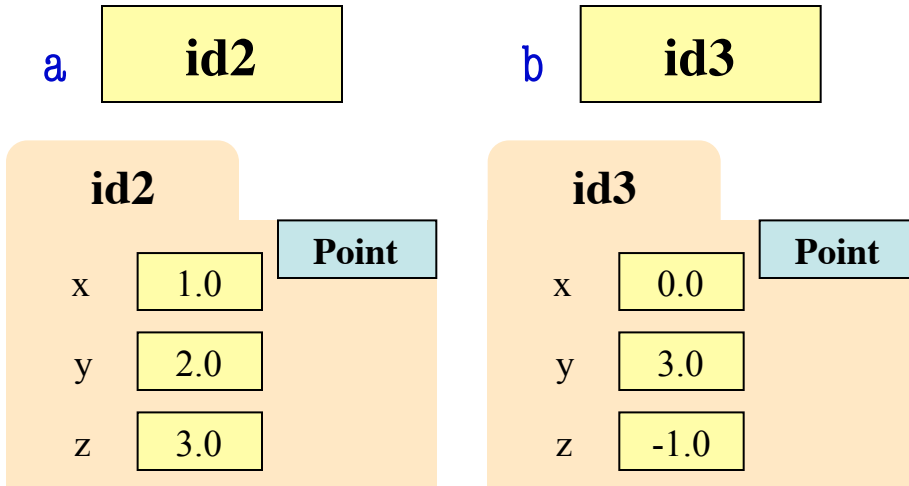
```
        assert type(q) == Point
```

```
        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 Point(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 Point"""
```

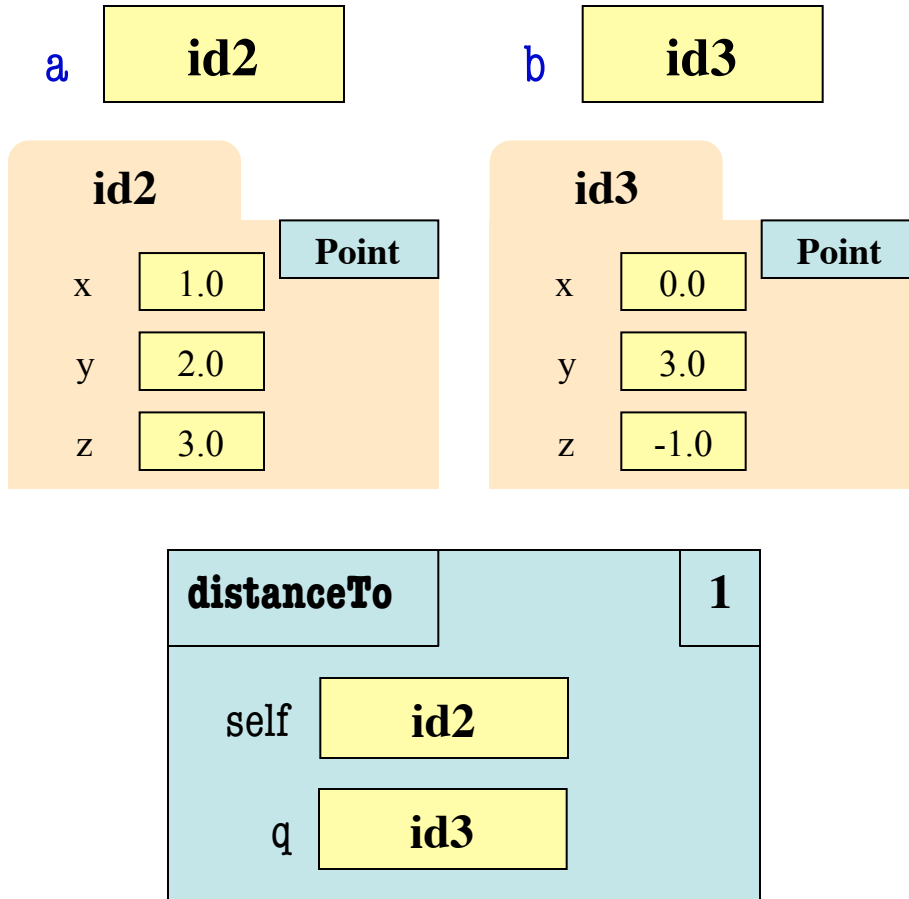
```
    assert type(q) == Point
```

```
    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 Point(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 Point"""
```

```
    assert type(q) == Point
```

```
    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)

---

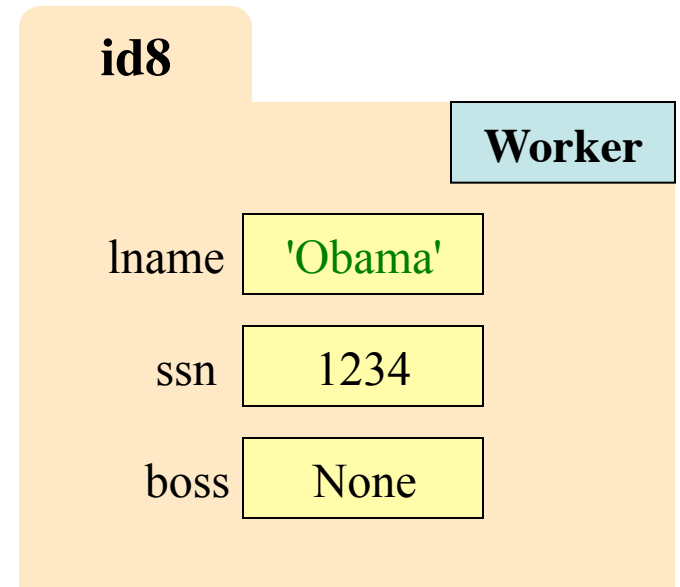
- Creating a new Worker is a multi-step process:
  - `w = Worker()` ← Instance is empty
  - `w.lname = 'White'`
  - ...
- Want to use something like
  - `w = Worker('White', 1234, None)`
  - Create a new Worker **and** assign attributes
  - lname to 'White', ssn to 1234, and boss to None
- Need a **custom constructor**

# Special Method: `__init__`

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

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

Called by the constructor



# Special Method: `__init__`

two underscores

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

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
```

Called by the constructor

id8

Worker

lname 'Obama'

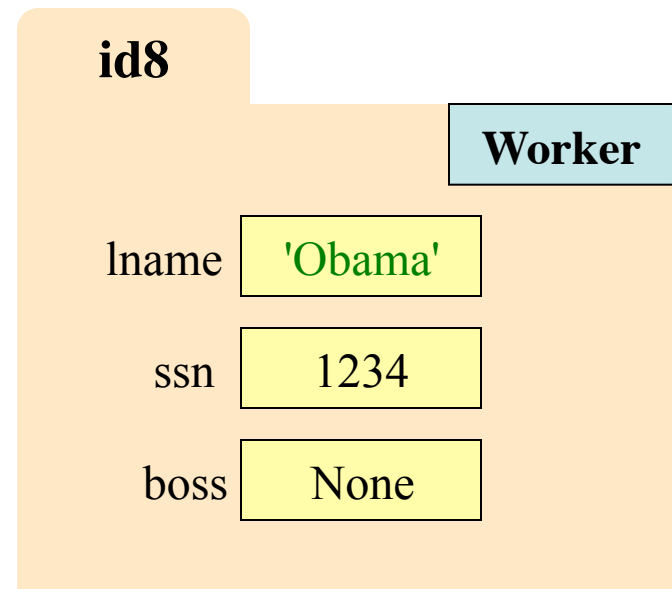
ssn 1234

boss None

# Evaluating a Constructor Expression

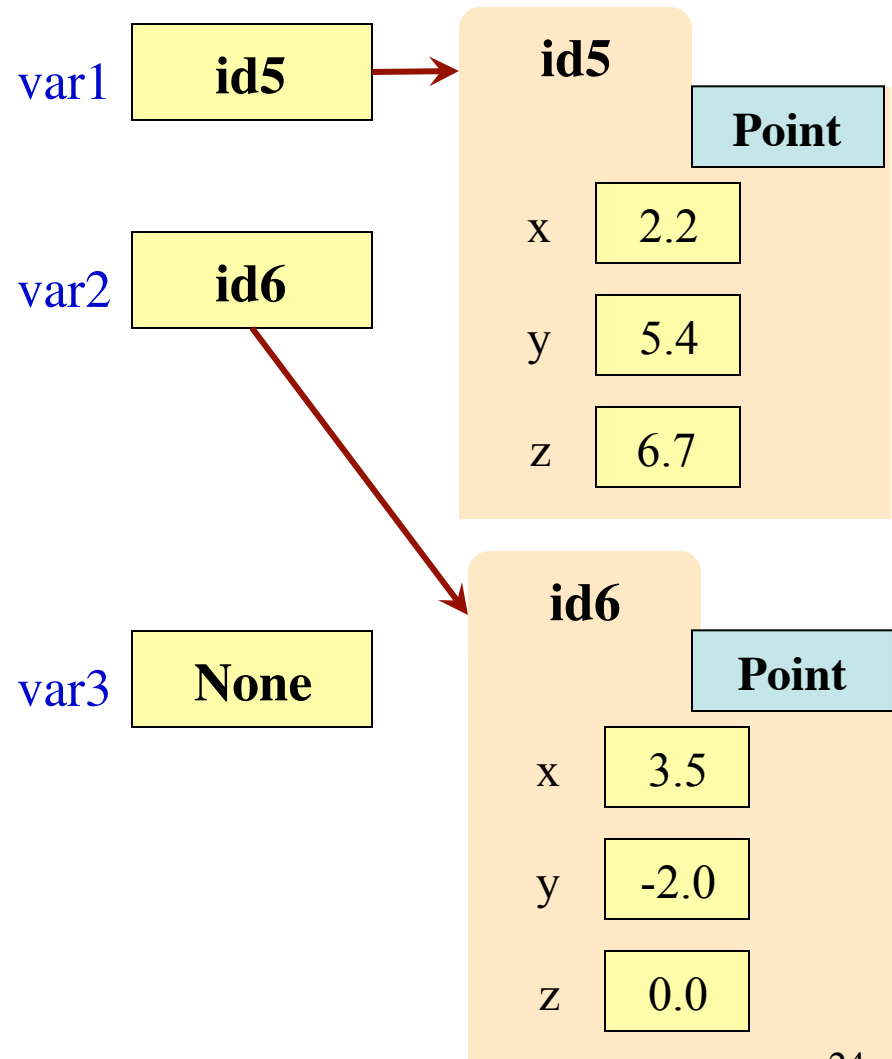
`Worker('Obama', 1234, None)`

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 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 (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 Point 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 = Point()`           # (0,0,0)
- `p = Point(1,2,3)`    # (1,2,3)
- `p = Point(1,2)`       # (1,2,0)
- `p = Point(y=3)`       # (0,3,0)
- `p = Point(1,z=2)`    # (1,0,2)

```
class Point(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 = Point()` # (0,0,0)
- `p = Point(1,2)` Assigns in order
- `p = Point(y=3)` Use parameter name when out of order
- `p = Point(1,z=2)` Can mix two approaches

```
class Point(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 = Point()` # (0,0,0)
- `p = Point(1,2)` Assigns in order
- `p = Point(y=3)` Use parameter name when out of order
- `p = Point(1,z=2)` Can mix two approaches

```
class Point(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 P
           Precondition:
```

Not limited to methods.  
Can do with any function.