

Lecture 15

Classes

Announcements for This Lecture

Reading

- Today: Chapters 15, 16
- Thursday: Chapter 17

Assignment 4

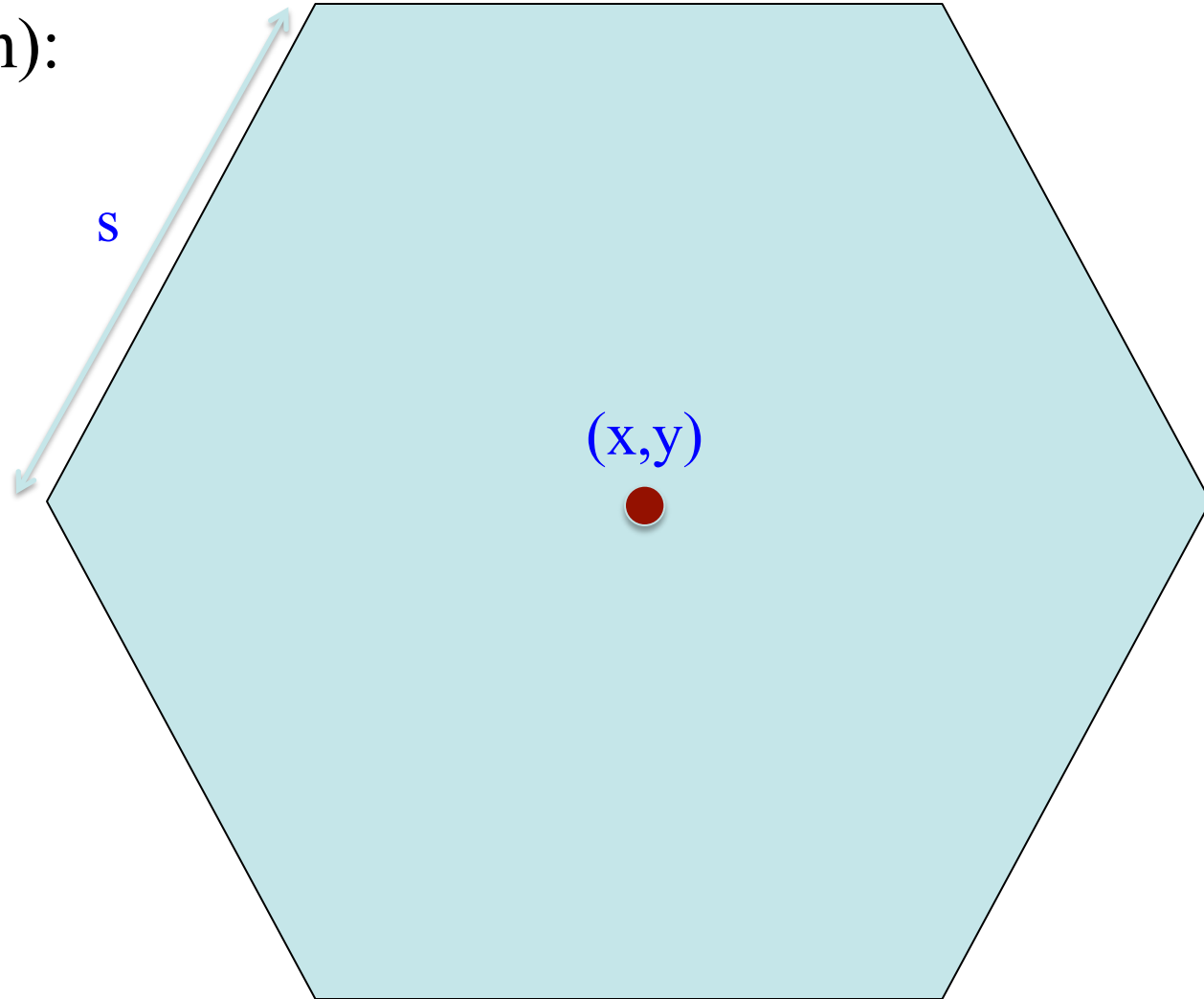
- Due week from Today
- Consultants on Saturday
- Next assignment short!
 - Only a week to work on it
 - Give more time for later 2

Grading

- Prelim 1 available
 - Get it in 305 Upson
 - Solution posted in CMS
 - See Piazza post on grades
 - **Mean: 86, Median: 89**
- A3 still being graded
 - About half are done
 - Posted before next class

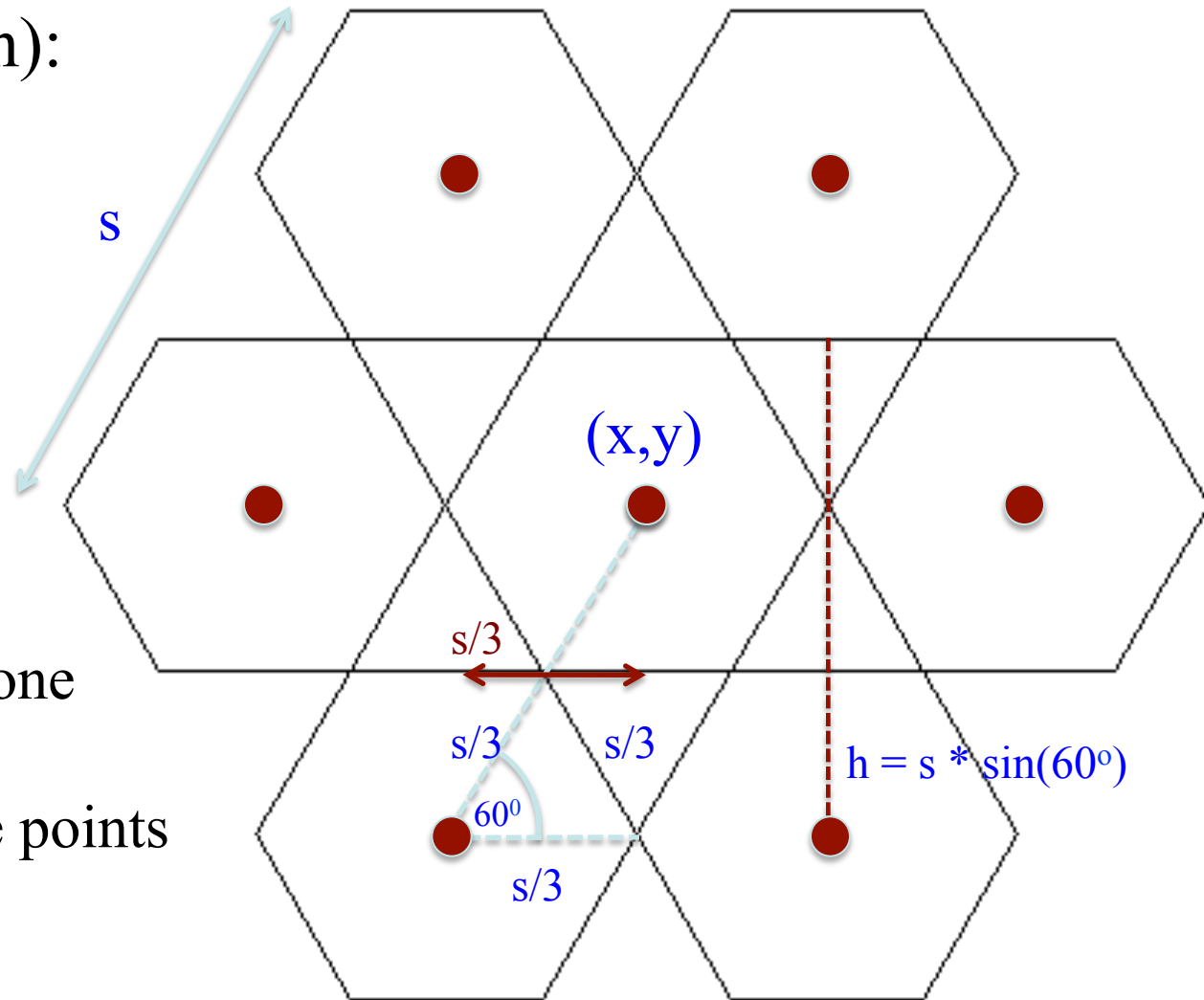
Grisly Snowflakes

- Given (as shown):
 - Length s
 - Point (x,y)



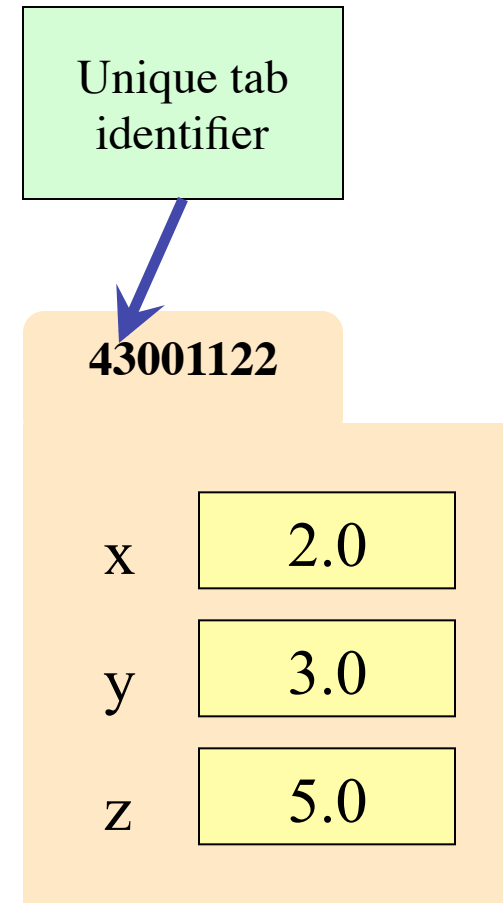
Grisly Snowflakes

- Given (as shown):
 - Length s
 - Point (x,y)
- Find:
 - Coordinates of all red points
- Draw:
 - Snowflakes of one less depth and size $s/3$ at those points



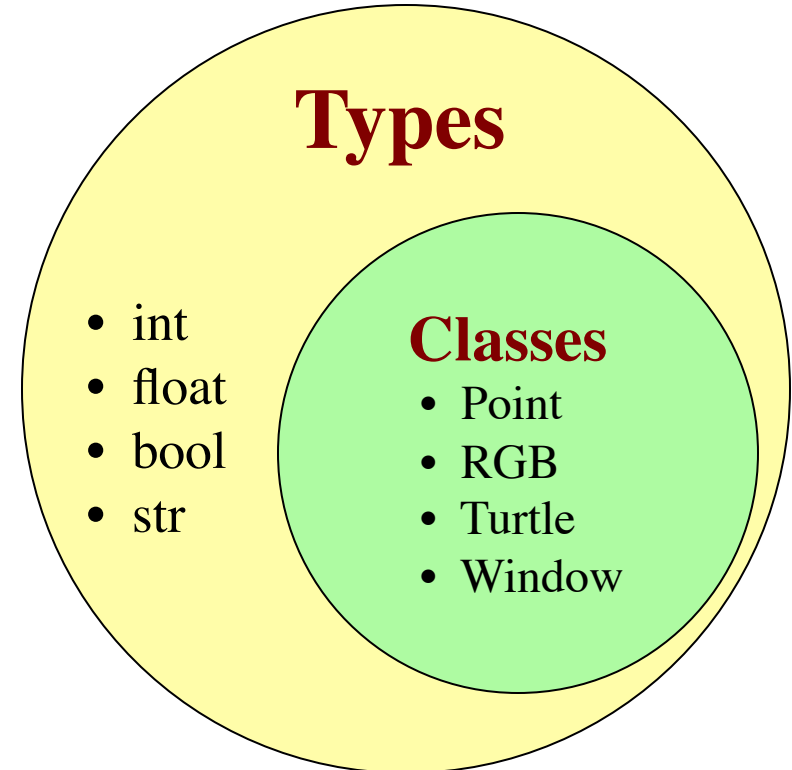
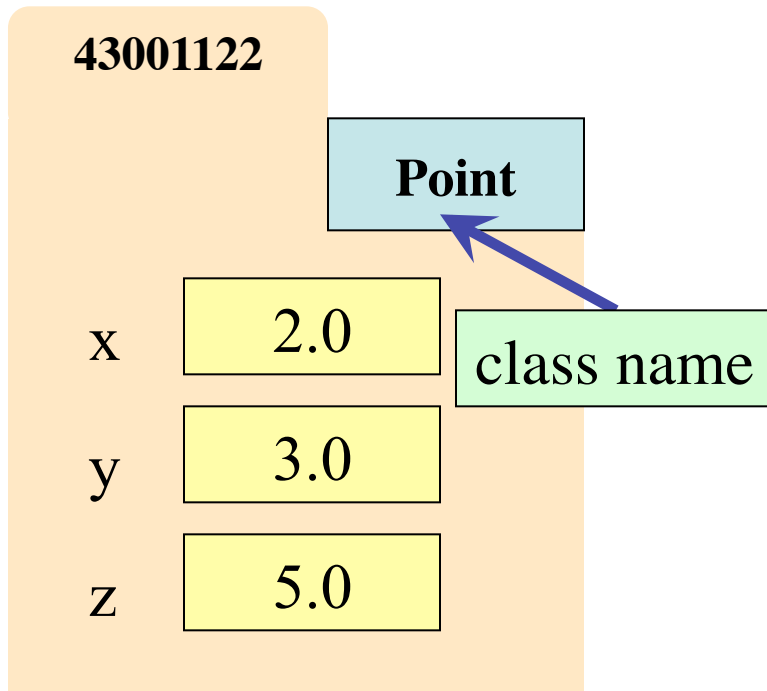
Recall: Objects as Data in Folders

- 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
 - You cannot ever change this



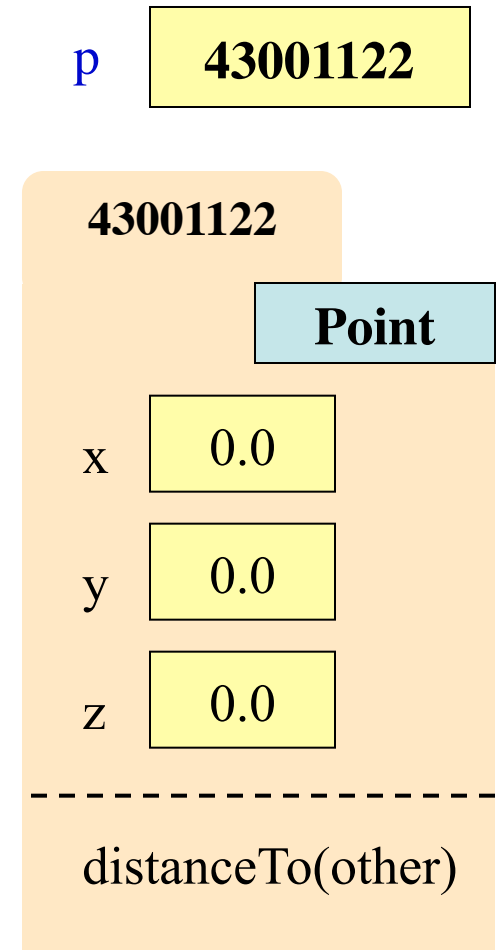
Recall: Classes are Types for Objects

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



Recall: Objects can have Methods

- **Method:** function tied to object
 - Has a function call part:
`<function-name>(<arguments>)`
 - But prefix it with variable name:
`<object-variable>.<function-call>`
 - Use of a method is a *method call*
- **Example:** `p.distanceTo(q)`
 - Both `p` and `q` act as arguments
 - Computes distance between two



The Class Definition

- Defines the format of any object of that class
- Everything is indented under the class name

class <class-name>(object):

"""Class specification"""

definitions of fields

definitions of methods

(in any order)

- Simplest class definition:

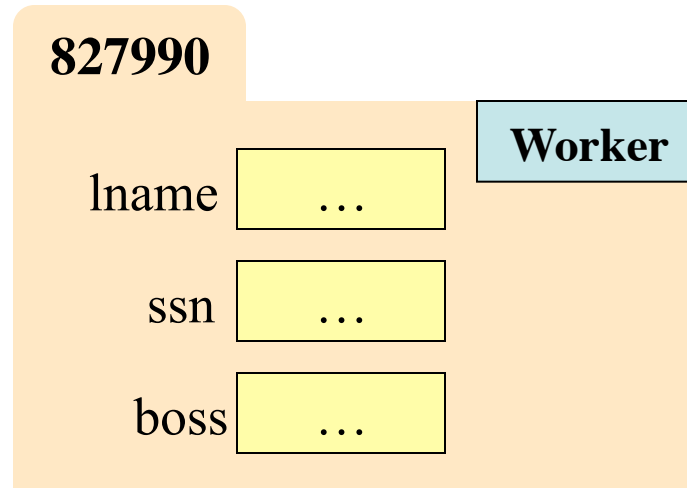
```
class Example(object):
```

```
    """Does nothing"""
```

```
    pass
```

This definition goes inside of a module, just like a function definition does.

Fields: Adding Attributes to a Class



```
class Worker(object):
```

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

```
    lname = "      # Last name (string, " if unknown)
```

```
    ssn = 0        # Social security # (int in range 0..9999999999)
```

```
    boss = None   # Immediate boss (Worker object; None if none)
```

Fields: Adding Attributes to a Class

827990

lname

...

ssn

...

boss

...

Worker

Default Values:
What new objects
all start off with

Invariants:
Properties that
are always true.

class Worker(object):

"""An instance is a worker in a certain organization."""

lname = "

Last name (string, " if unknown)

ssn = 0

Social security # (int in range 0..999999999)

boss = None

Immediate boss (Worker object; None if none)

Fields: Adding Attributes to a Class

827990

lname

...

ssn

...

boss

...

Worker

Default Values:
What new objects
all start off with

Invariants:
Properties that
are always true.

```
class Worker(object):
```

```
    """An instance is a worker in a certain organization"""
```

```
    lname = "
```

```
    ssn = 0
```

```
    boss = None
```

```
    # Last name (string)
```

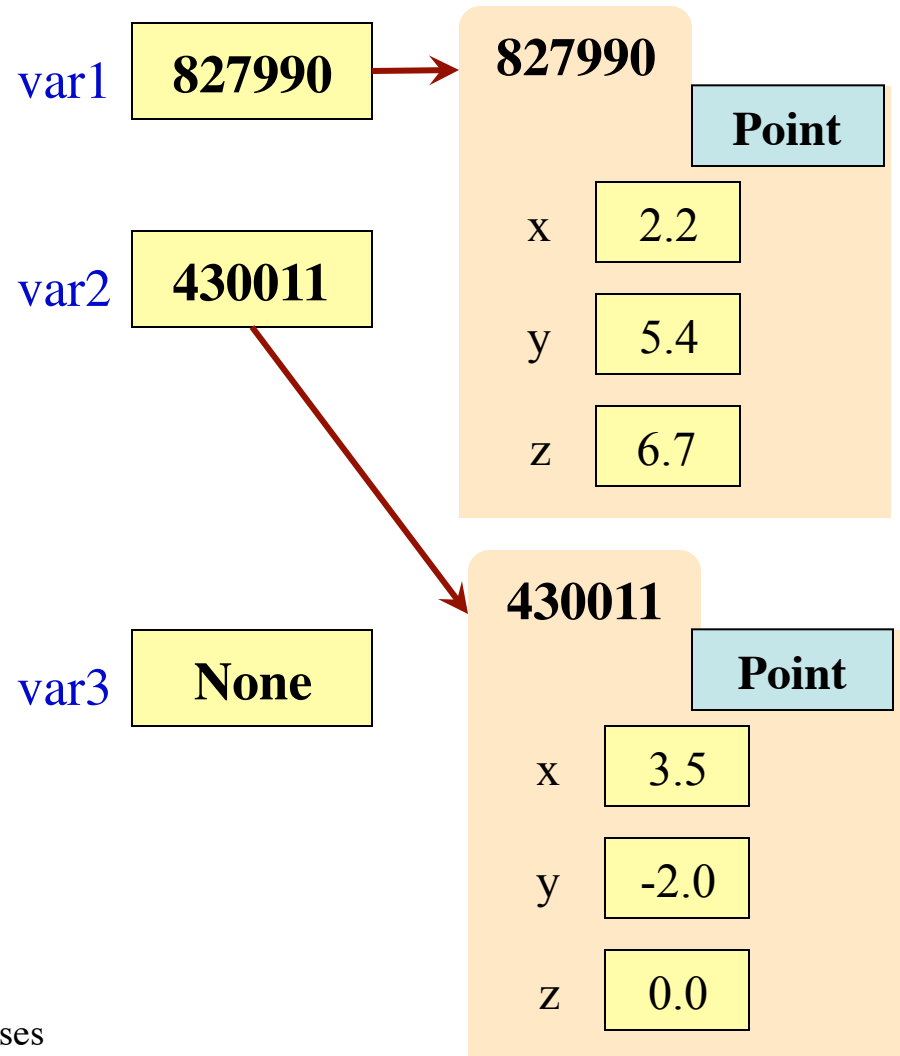
```
    # Social Security Number (int)
```

```
    # In charge (object; None if none)
```

Right now, invariants are comments.
We will enforce them later.

The Value None

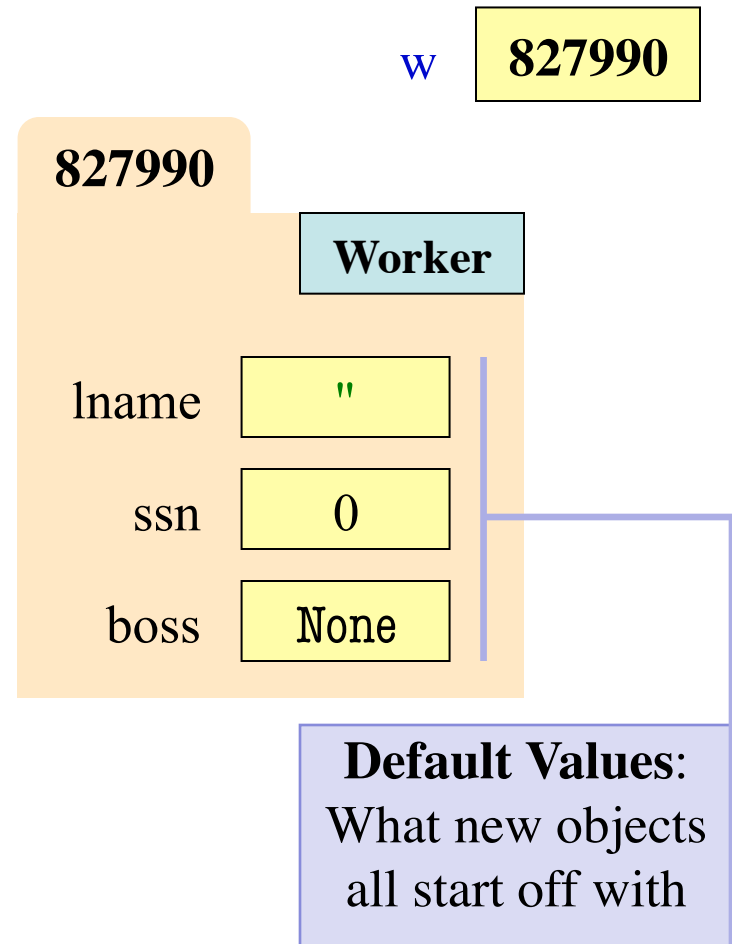
- The boss field is a problem.
 - boss is a Worker object
 - But we are defining what the Worker class looks like
 - **Cannot put a value in boss until the definition is done**
- **Solution:** use value None
 - **None:** Lack of (folder) name
 - Will reassign the field later!
- Be careful with None variables
 - `var3.x` gives error!
 - There is no name in var3
 - Which Point to use?



Constructors

- Class definition creates a special (hidden) function
 - **Same name as the class**
 - (For now) no arguments
- Called the **constructor**
 - Makes new object of class
 - Returns the id of object
- Example:
 - Point()
 - Worker()

w = Worker()



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 Point(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 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)`

a **827990**

b **430011**

827990

x 1.0

y 2.0

z 3.0

Point

430011

x 0.0

y 3.0

z -1.0

Point

```
class Point(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 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 +
```

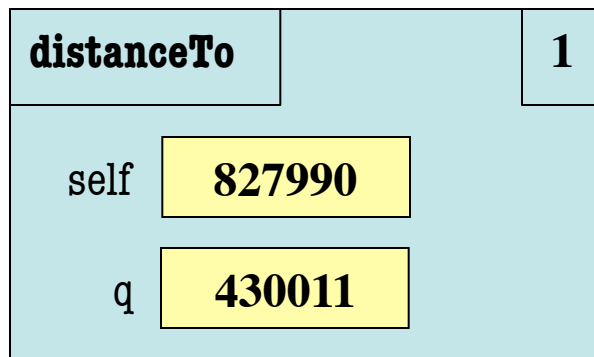
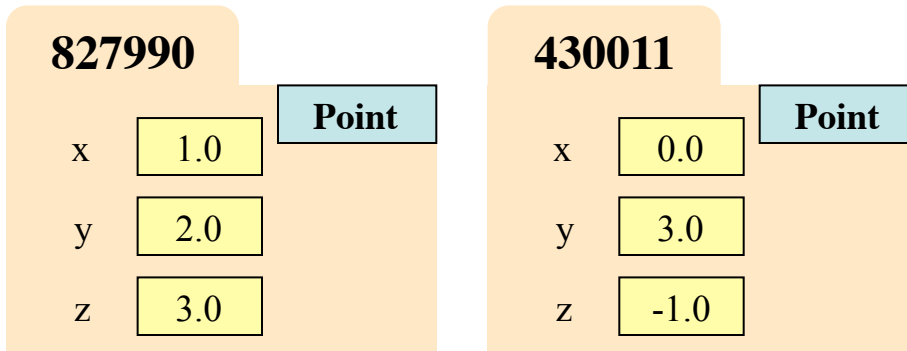
```
              (self.z-q.z)**2)
```

```
    return math.sqrt(sqrdst)
```

Methods Calls

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

a **827990** b **430011**



```
class Point(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 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 Fields of an Object (Folder)

- Creating a new Worker is a multi-step process:

- `w = new Worker()`

← Fields are all defaults

- `w.lname = 'White'`

- ...

- Want to use something like

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

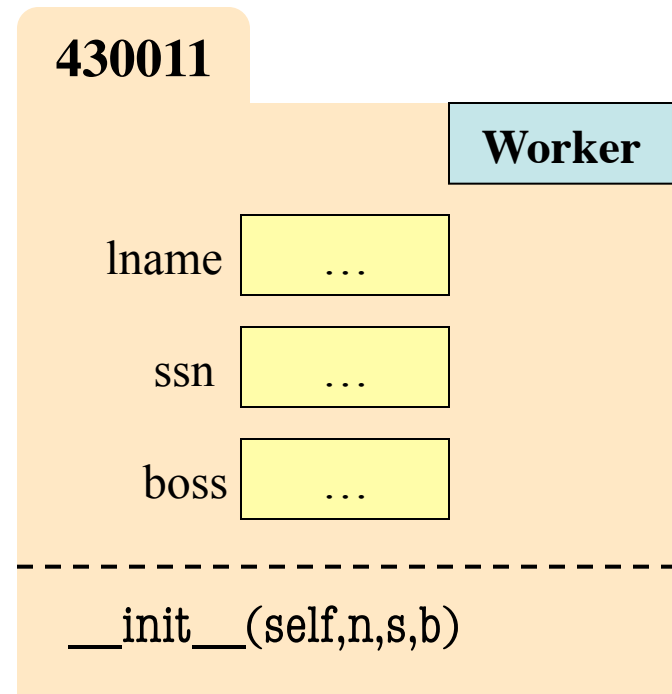
- Create a new Worker **and** assign fields

- lname to 'White', ssn to 1234, and boss to None

- Need a **custom constructor**

Special Method: `__init__`

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

two underscores

don't forget self

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

"""Constructor: creates a Worker

Instance 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

430011

Worker

lname ...

ssn ...

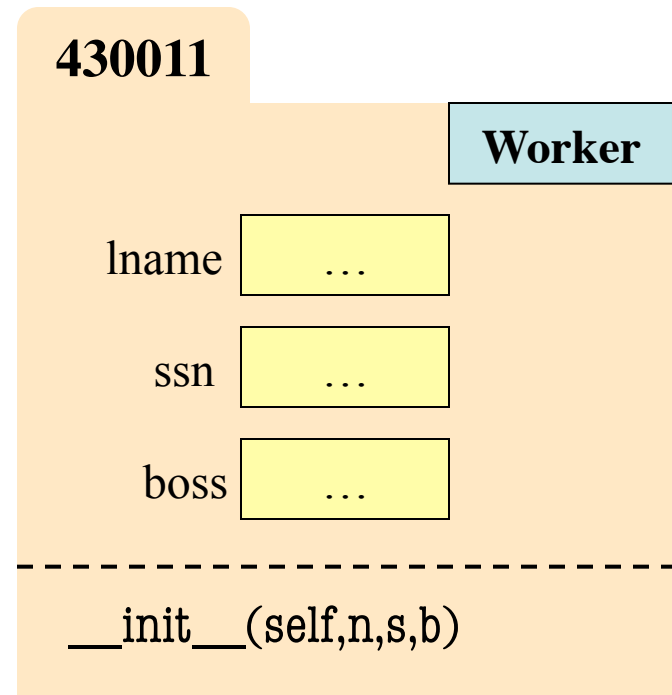
boss ...

`__init__(self,n,s,b)`

How a Constructor Expression Works

`Worker('White', 1, None)`

1. Creates a new object (folder) of the class `Worker`
 - Fields set to default values
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 as final value of expression



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 = 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 = 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 = 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 = 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 = 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
        Precondition: ..."""
```

Not limited to methods.
Can do with any function.

What Does `str()` Do On Objects?

- Does **NOT** display contents

```
>>> p = Point(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 Point(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 = Point(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 Point(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 Point(object):

```
"""Instances are 3d points"""
```

```
x = 0.0 # x coord, float
```

```
y = 0.0 # y coord, float
```

```
z = 0.0 # z coord, float
```

```
...
```

3.0-Style Classes
Well-Designed

NO

class Point:

```
"""Instances are 3d points"""
```

```
x = 0.0 # x coord, float
```

```
y = 0.0 # y coord, float
```

```
z = 0.0 # z coord, float
```

```
...
```

“Old-Style” Classes
Very, Very Bad