Lecture 18

Classes

Announcements for This Lecture

Assignments

- **A4** **Friday** at midnight
  - Hopefully you are on Task 4
  - That and task 5 are hardest
- Will post **A5** on Saturday
  - Written assignment like A2
  - Needs material from next Tues
- Will post **A6** on Monday.
  - Not due until November 14
  - But it is **much longer**!

Optional Videos

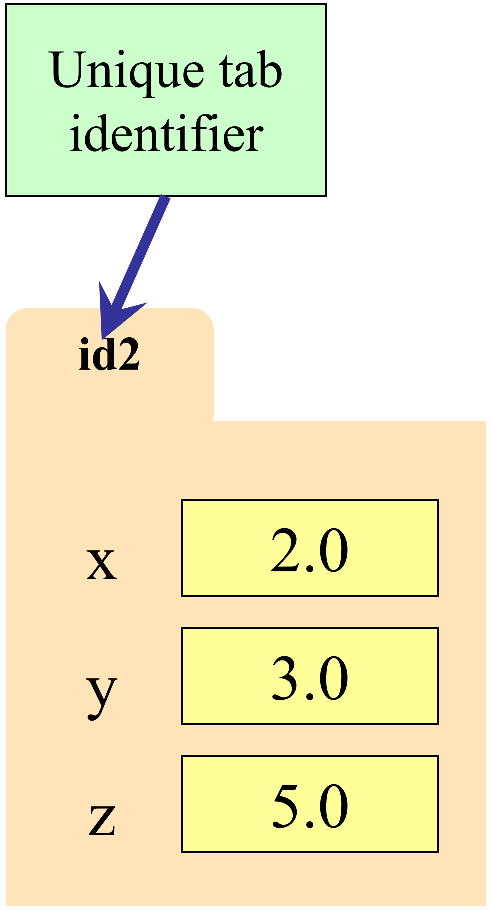
- **Videos 20.1-20.8** today
- **Videos 20.9-20.10** next time
- Also **Lesson 21** next time

Exams

- Last week for regrades
  - Limit them to valid issues
- We will do them *eventually*

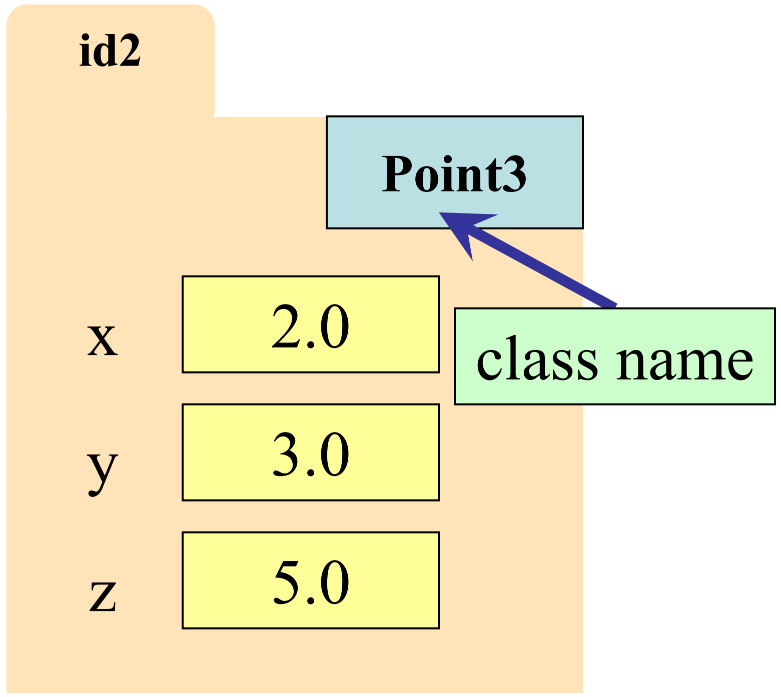
Recall: Objects as Data in Folders

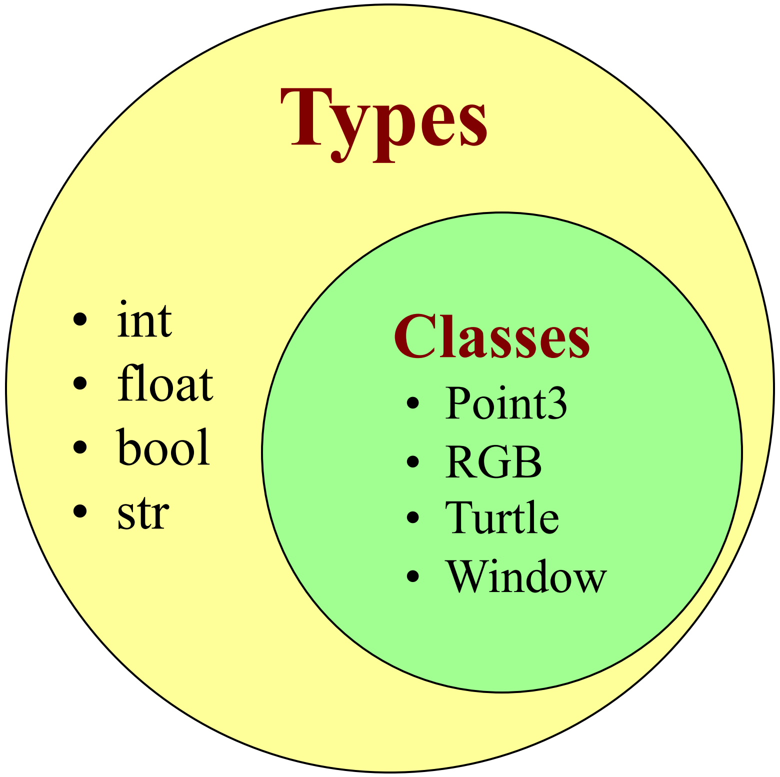
- An object is like a **manila folder**
- It contains other variables
  - Variables are called **attributes**
  - Can change values of an attribute (with assignment statements)
- It has a "tab" that identifies it
  - Unique number assigned by Python
  - Fixed for lifetime of the object



Recall: Classes are Types for Objects

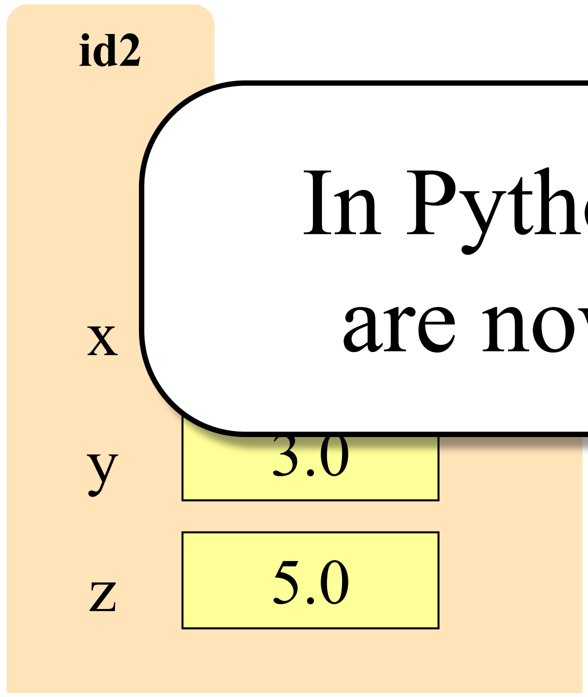
- Values must have a type
  - An **object** is a **value**
  - A **class** is its **type**
- Classes are how we add new types to Python

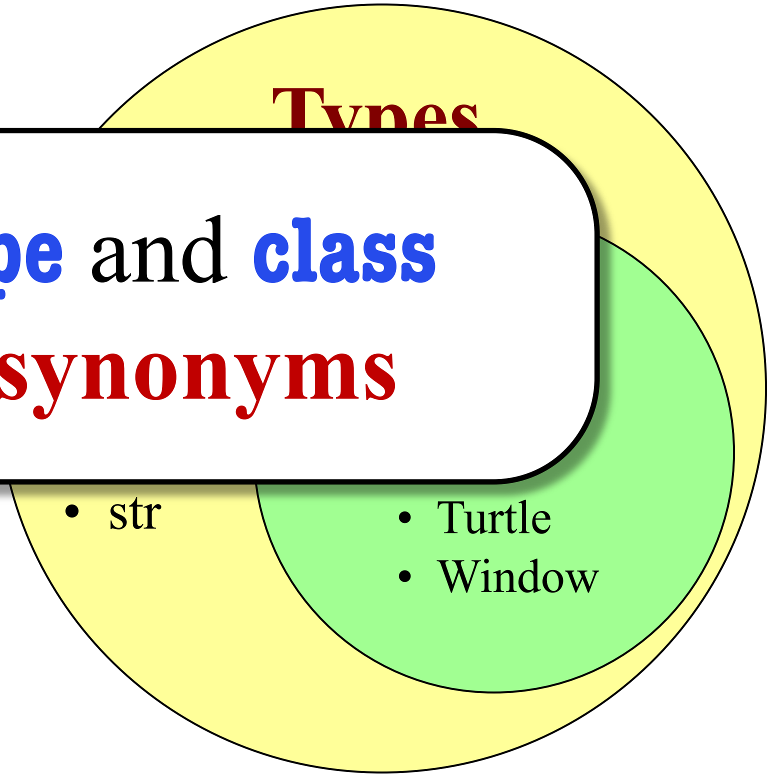




Recall: Classes are Types for Objects

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



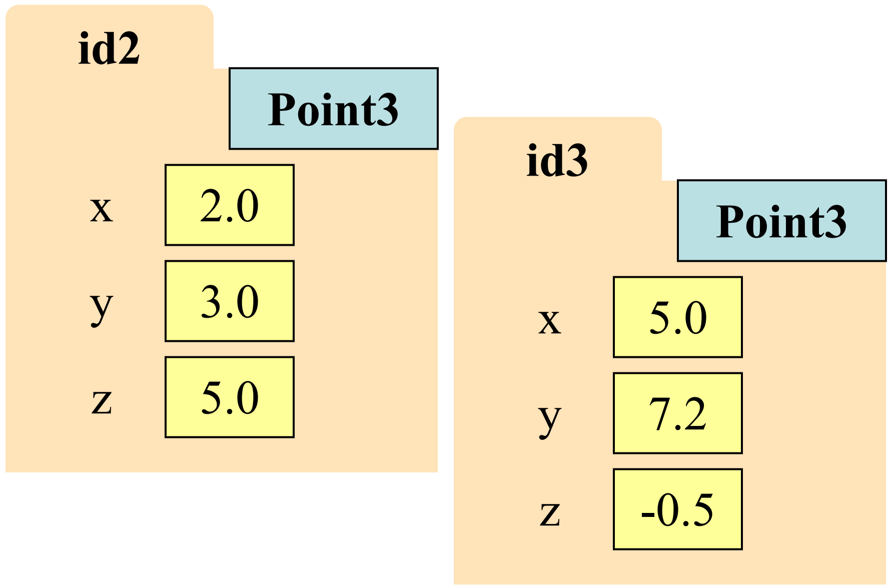


In Python3, **type** and **class** are now both **synonyms**

Classes Have Folders Too

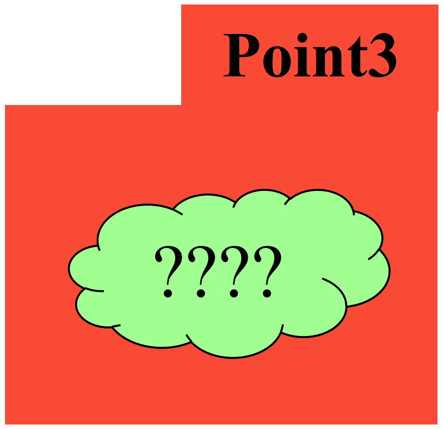
Object Folders

- Separate for each *instance*



Class Folders

- Data common to all instances



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):
    """Class specification"""
    <function definitions>
    <assignment statements>
    <any other statements also allowed>
```

Do not forget the colon!

more on this later

Specification (similar to one for a function)

to define **methods**

to define **attributes**

…but not often used

Example

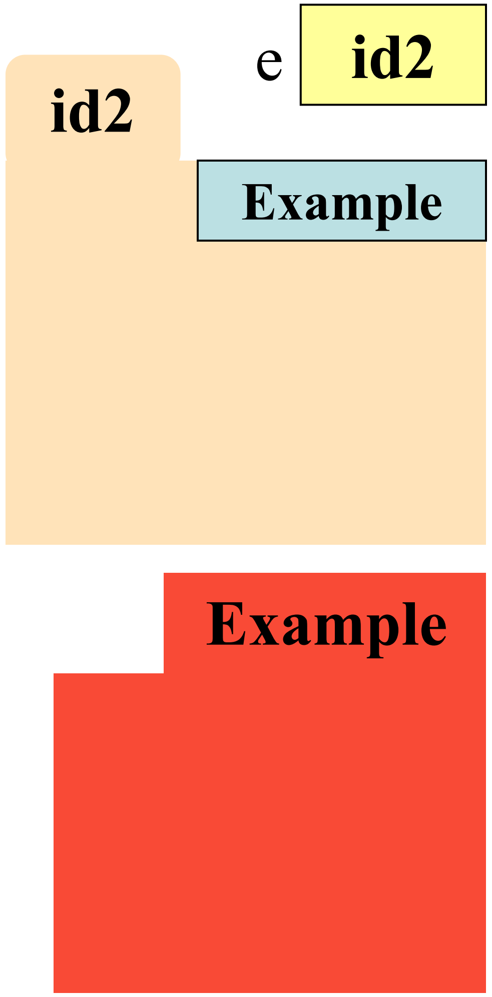
```
class Example(object):
    """The simplest possible class."""
    pass
```

Python creates after reading the class definition

Recall: Constructors

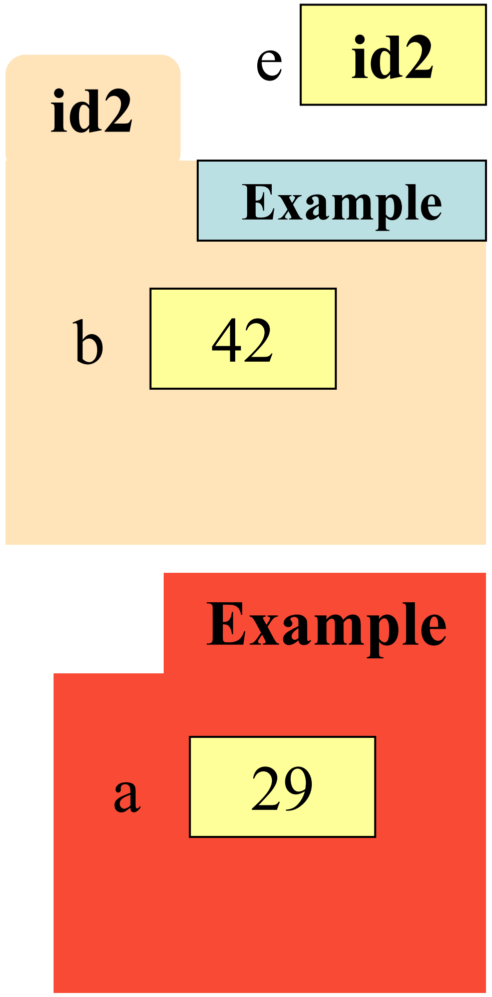
- Function to create new instances
  - Function name == class name
  - Created for you automatically
- Calling the constructor:
  - Makes a new object folder
  - Initializes attributes
  - Returns the id of the folder
- By default, takes no arguments
  - `e = Example()`

Will come back to this



Instances and Attributes

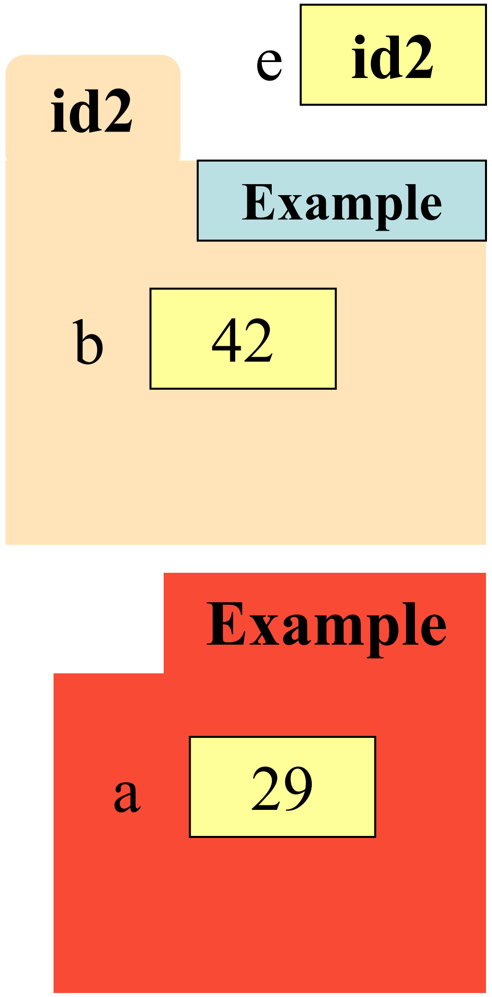
- Assignments add object attributes
  - <object>.<att> = <expression>
  - **Example**: `e.b = 42`
- Assignments can add class attributes
  - <class>.<att> = <expression>
  - **Example**: `Example.a = 29`
- Objects can access class attributes
  - **Example**: `print(e.a)`
  - But assigning it creates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



Instances and Attributes

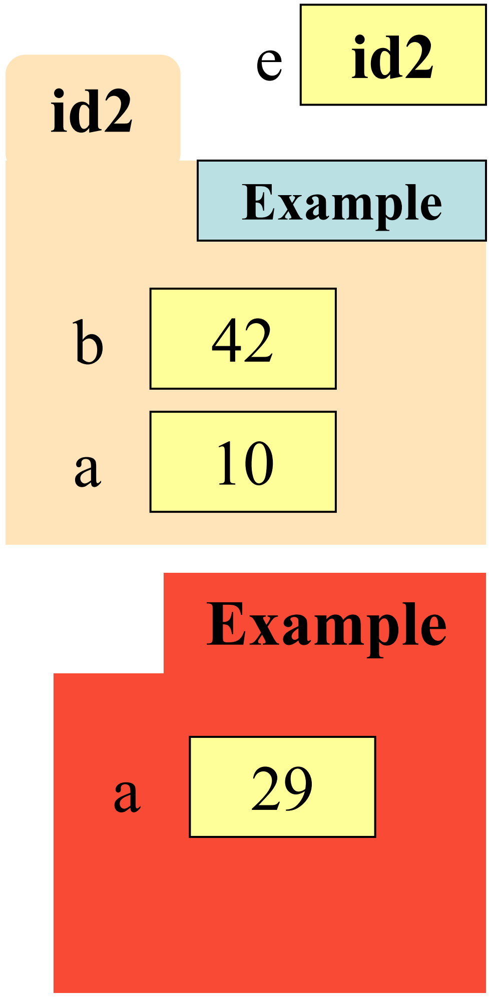
- Assignments add object attributes
  - <object>.<att> = <expression>
  - **Example**: `e.b = 42`
- Assignments can add class attributes
  - <class>.<att> = <expression>
  - **Example**: `Example.a = 29`
- Objects can access class attributes
  - **Example**: `print(e.a)`
  - But assigning it creates 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**:
  - 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):
    """A class representing a worker in a certain organization

     Instance has basic worker info, but no salary information.

    Attribute lname: The worker last name
    Invariant: lname is a string

    Attribute ssn: The Social Security number
    Invariant: ssn is an int in the range 0..999999999

    Attribute boss: The worker's boss
    Invariant: boss is an instace of Worker, or None if no boss"""
```

The Class Specification

```
class Worker(object):
    """A class representing a worker in a certain organization

     Instance has basic worker info, but no salary information.

    Attribute lname: The worker last name
    Invariant: lname is a string

    Attribute ssn: The Social Security number
    Invariant: ssn is an int in the range 0..999999999

    Attribute boss: The worker's boss
    Invariant: boss is an instace of Worker, or None if no boss"""
```

Short summary

More detail

Description

Invariant

The Class Specification

```
class Worker(object):
    """A class representing a worker in a certain organization

     Instance has basic worker info. but no salarv information.

    Attribu
    Invaria

    Attribute ssn: The Social Security number
    Invariant: ssn is an int in the range 0..999999999

    Attribute boss: The worker's boss
    Invariant: boss is an instace of Worker, or None if no boss"""
```

Warning: New format since 2019.
2018 exams will be very different.

Recall: Objects can have Methods

- Object before the name is an *implicit* argument
- **Example**: distance

```
>>> p = Point3(0,0,0)      # First point
>>> q = Point3(1,0,0)      # Second point
>>> r = Point3(0,0,1)      # Third point
>>> p.distance(r)          # Distance between p, r
1.0
>>> q.distance(r)          # Distance between q, r
1.4142135623730951
```

Method Definitions

- Looks like a function def
  - Indented *inside* class
  - First param is always self
  - But otherwise the same
- In a **method call**:
  - One less argument in ()
  - Obj in front goes to self
- **Example**: a.distance(b)

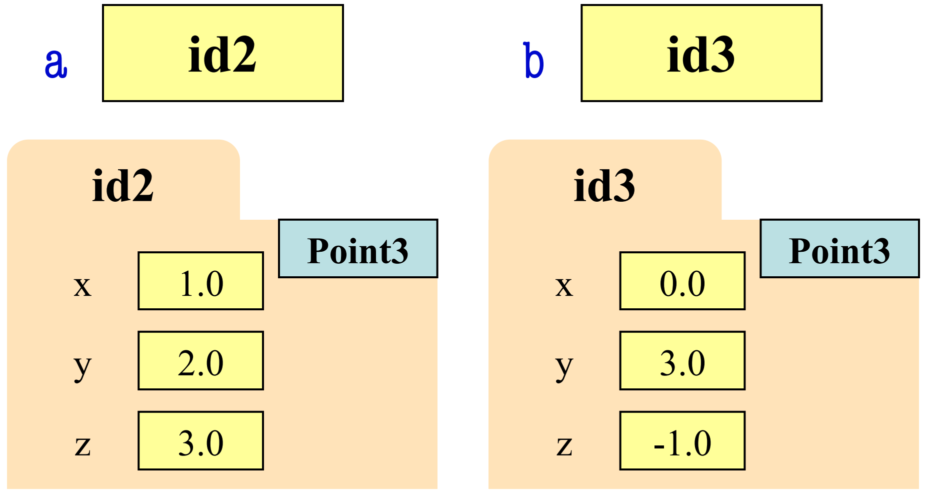
self

q

```
class Point3(object):
    """Class for points in 3d space
    Invariant: x is a float
    Invariant y is a float
    Invariant z is a float    """
    def distance(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

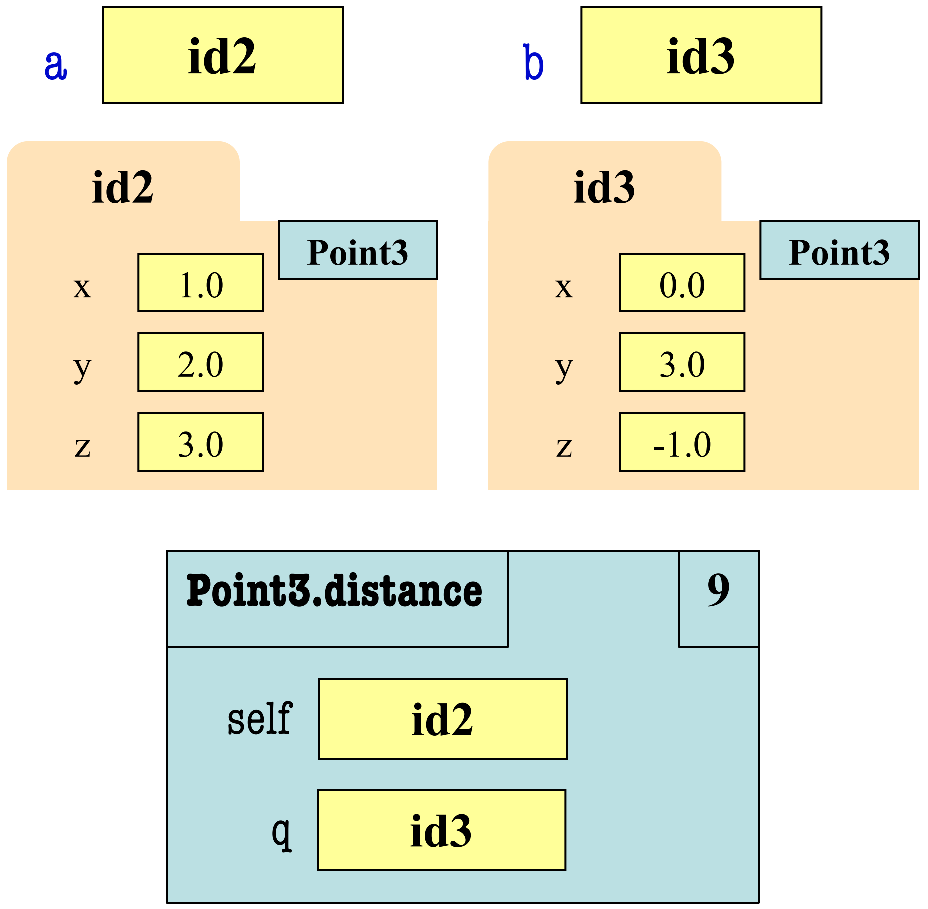
- **Example**: a.distance(b)



```
class Point3(object):
    """Class for points in 3d space
    Invariant: x is a float
    Invariant y is a float
    Invariant z is a float    """
    def distance(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.distance(b)

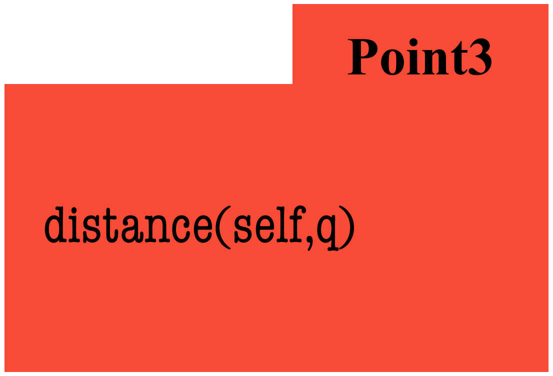


```
1.  class Point3(object):
2.      """Class for points in 3d space
3.      Invariant: x is a float
4.      Invariant y is a float
5.      Invariant z is a float     """
6.      def distance(self,q):
7.          """Returns dist from self to q
8.          Precondition: q a Point3"""
9.          assert type(q) == Point3
10.         sqrdst = ((self.x-q.x)**2 +
11.                   (self.y-q.y)**2 +
12.                   (self.z-q.z)**2)
13.         return math.sqrt(sqrdst)
```

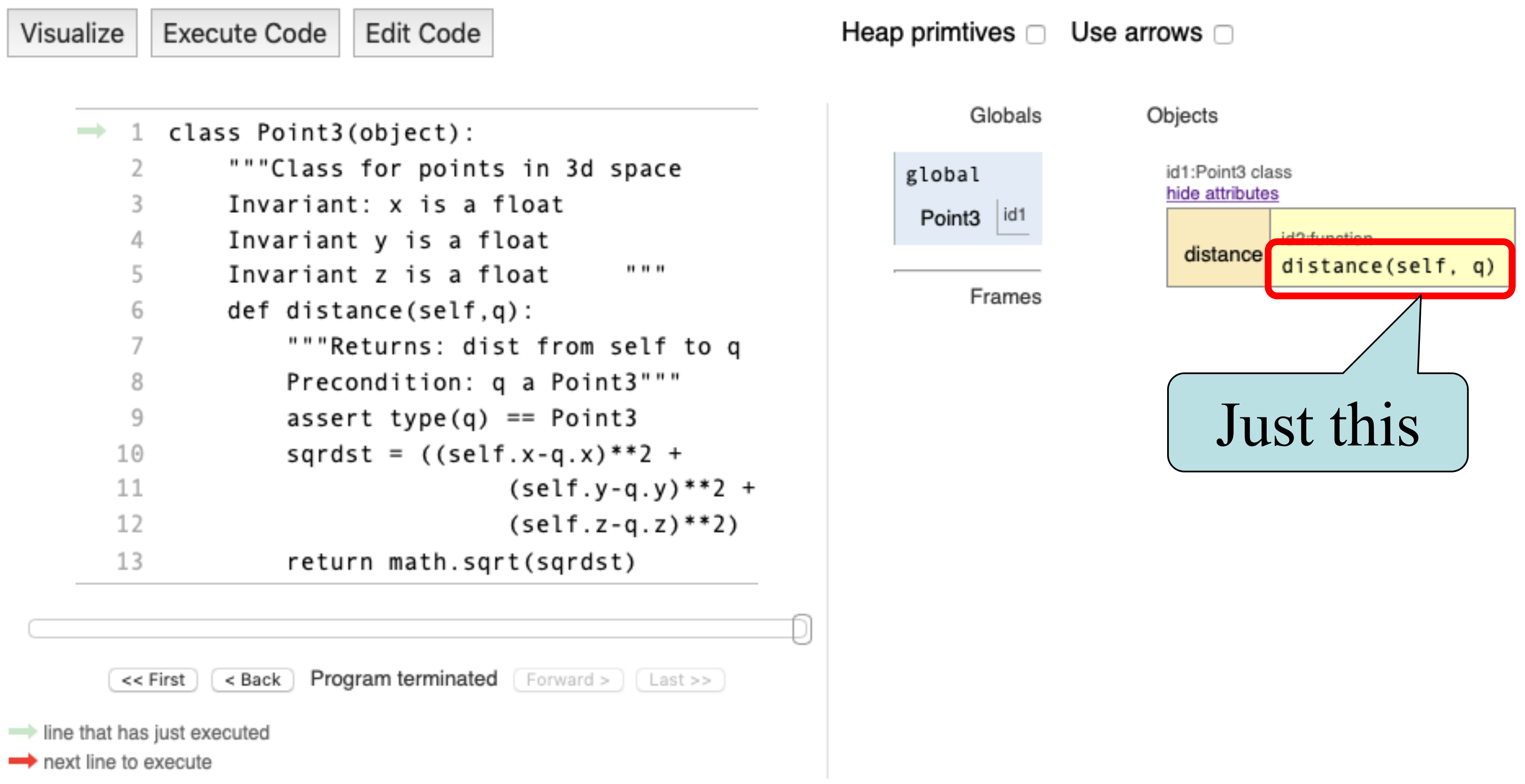
Methods and Folders

- Function definitions…
  - make a folder in heap
  - assign name as variable
  - variable in global space
- Methods are similar...
  - Variable in **class folder**
  - But otherwise the same
- **Rule of this course**
  - Put header in class folder
  - Nothing else!

```
class Point3(object):
    """Class for points in 3d space
    Invariant: x is a float
    Invariant y is a float
    Invariant z is a float     """
    def distance(self,q):
    ....
```



Methods and Folders



```python
class Point3(object):
    """Class for points in 3d space
    Invariant: x is a float
    Invariant y is a float
    Invariant z is a float     """
    def distance(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)
```

Just this

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)
```

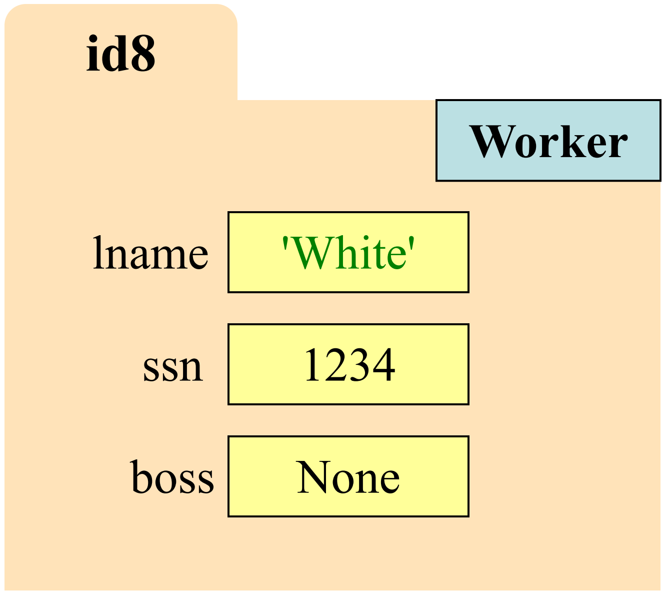
  - Create a new Worker **and** assign attributes
  - lname to 'White', ssn to 1234, and boss to None
- Need a **custom constructor**

Special Method: __init__

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

```
def  __init__(self, n, s, b):
    """Initializes a Worker object

    Has last name n, SSN s, and boss b

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

Called by the constructor



Special Method: __init__

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

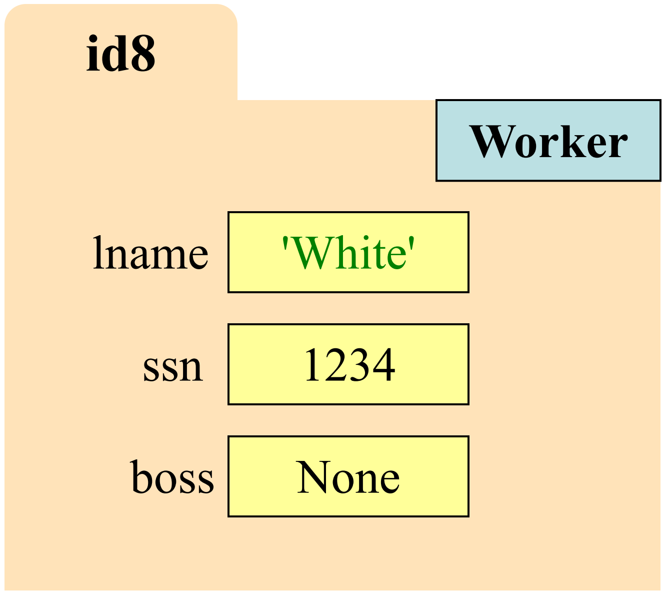
two underscores

don't forget self

```
def __init__(self, n, s, b):
    """Initializes a Worker object

    Has last name n, SSN s, and boss b

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

use self to assign attributes

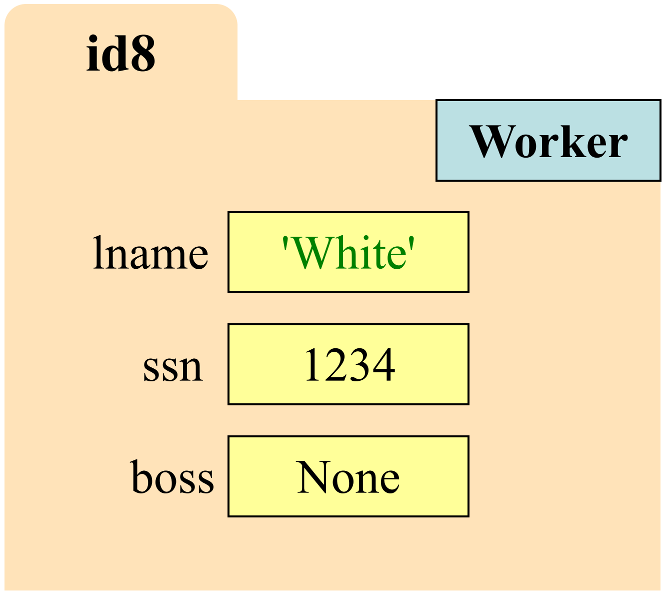
Called by the constructor



Evaluating a Constructor Expression

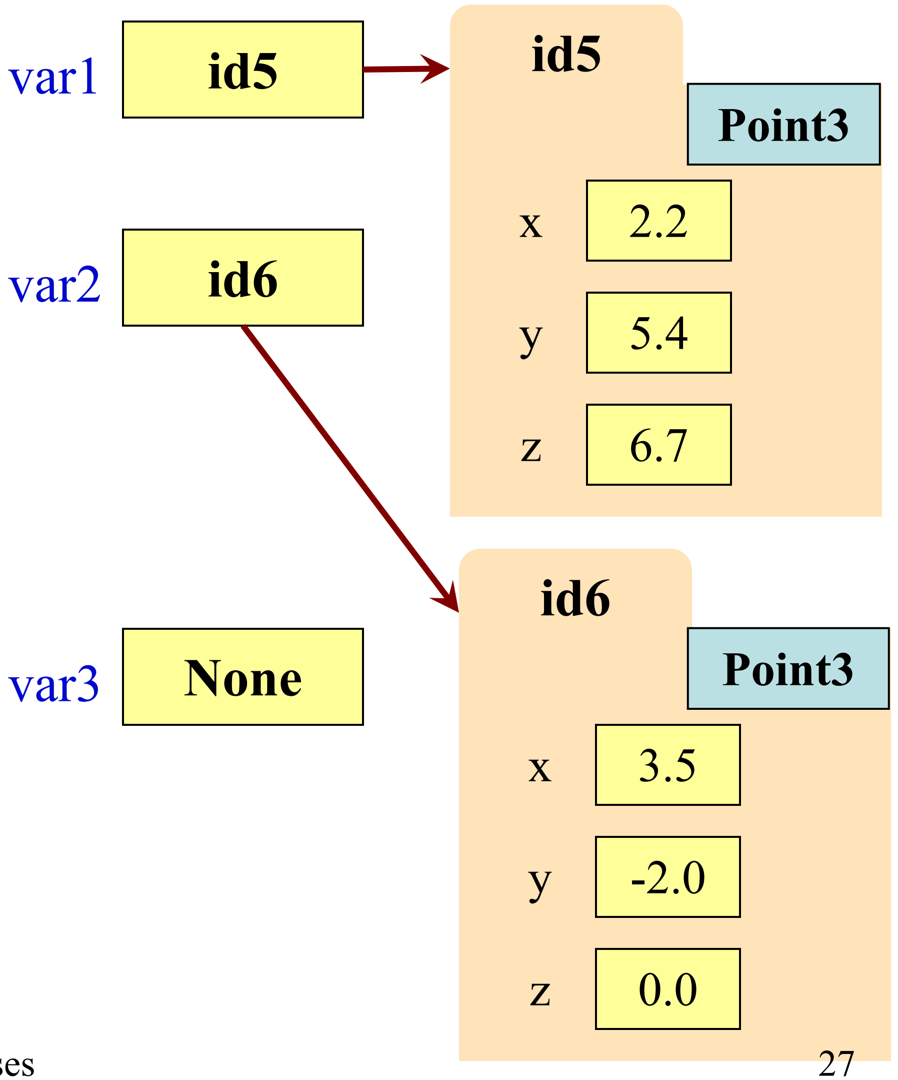
`Worker('White', 1234, None)`

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 Point3 to use?



A Class Definition

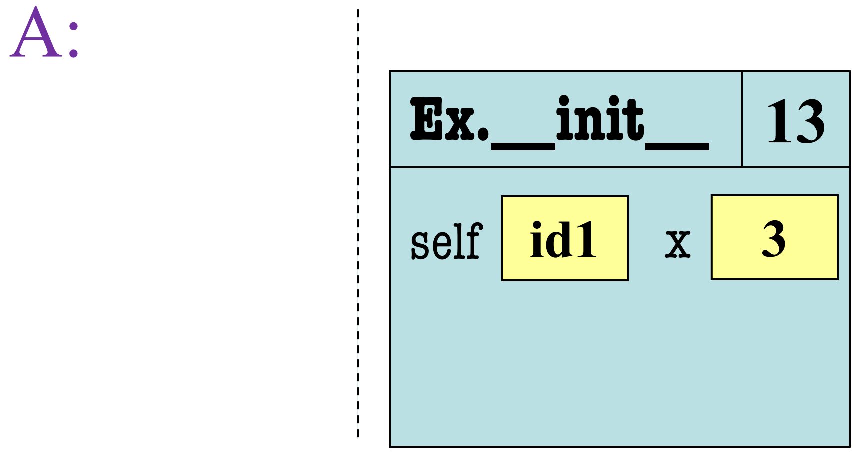
```
class Example(object):

    def __init__(self,x):
        self.x = x

    def foo(self,y):
        x = self.bar(y+1)
        return x

    def bar(self,y):
        self.x = y-1
        return self.x
```

```
>>> a = Example(3)
```

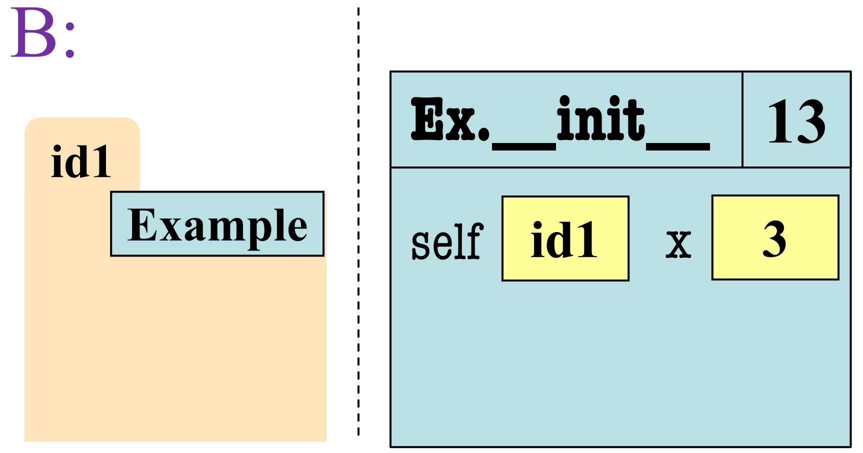
Ignoring the **class folder** what does the **call stack** and the **heap** look like?

Which One is Closest to Your Answer?

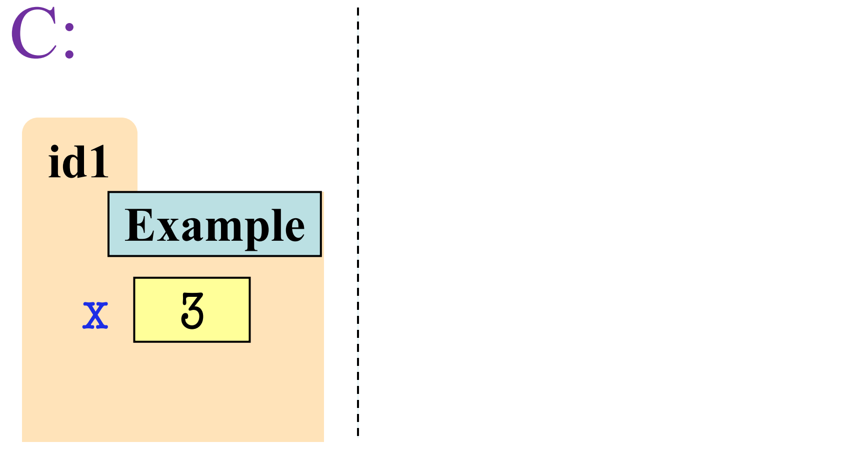
A:



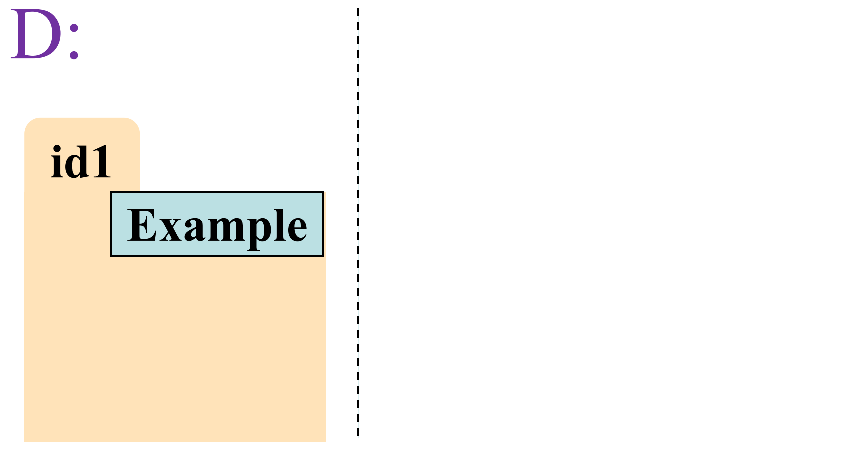
B:



C:



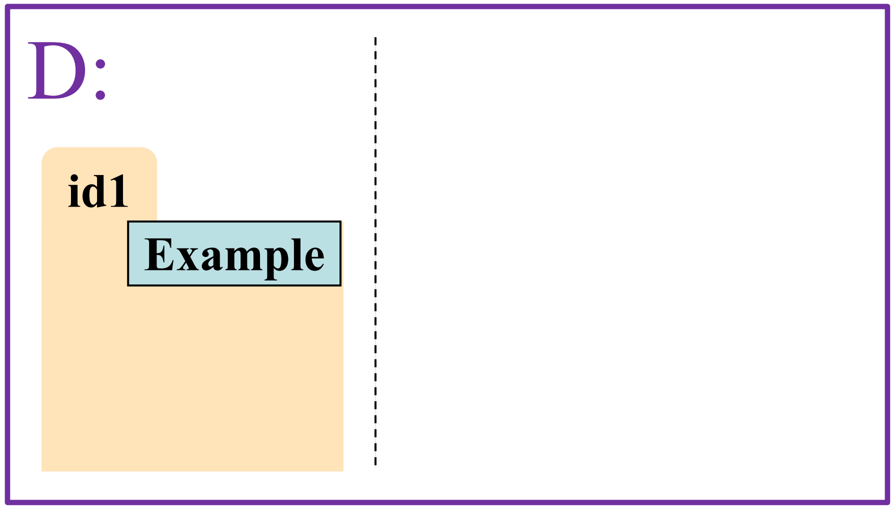
D:



A Class Definition

```
class Example(object):

    def __init__(self,x):
        self.x = x

    def foo(self,y):
        x = self.bar(y+1)
        return x

    def bar(self,y):
        self.x = y-1
        return self.x
```

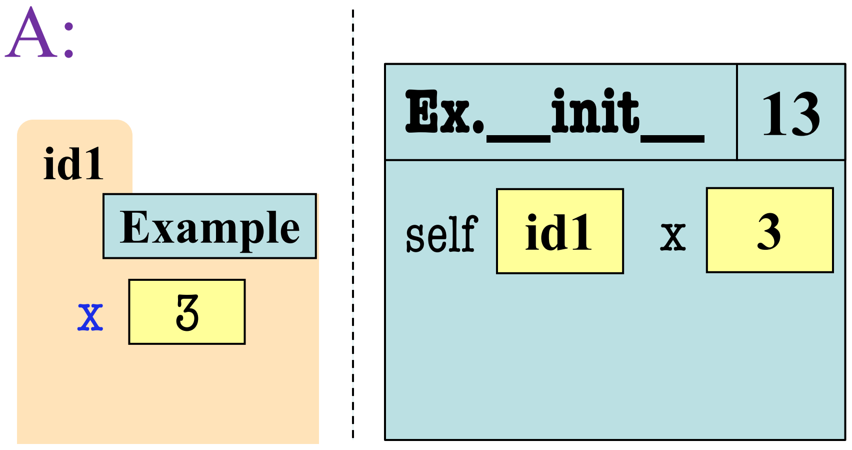
```
>>> a = Example(3)
```



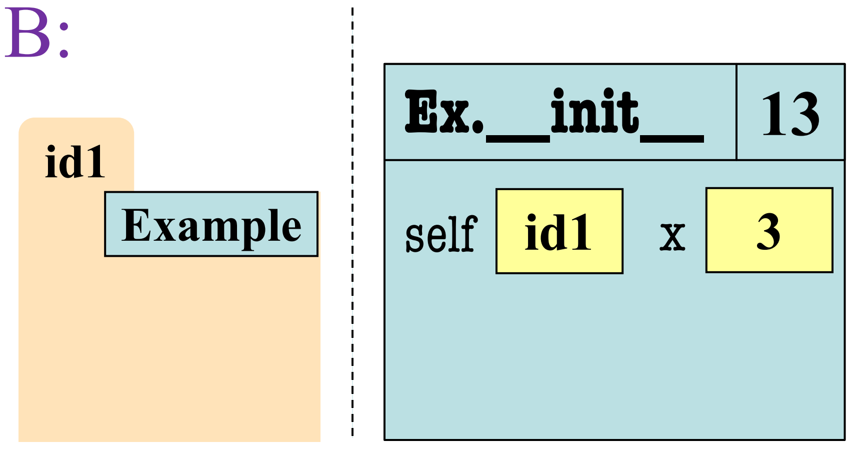
What is the **next step**?

Which One is Closest to Your Answer?

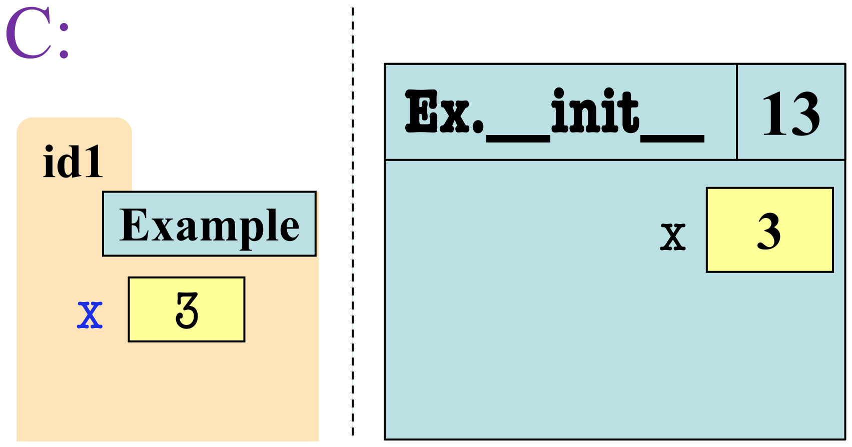
A:



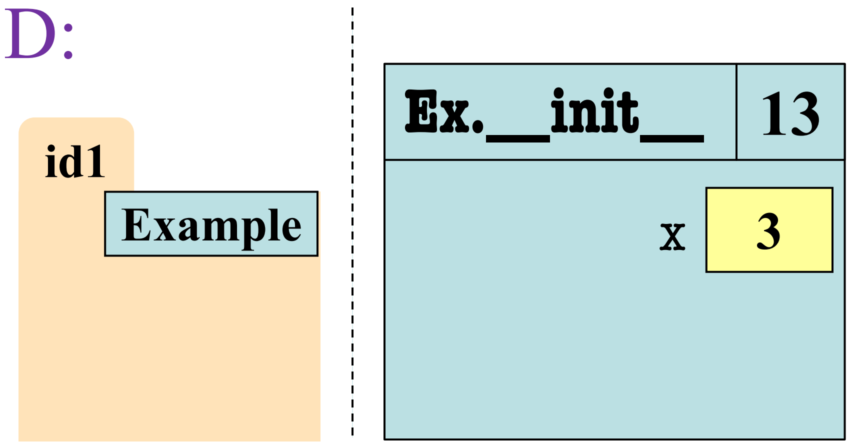
B:



C:



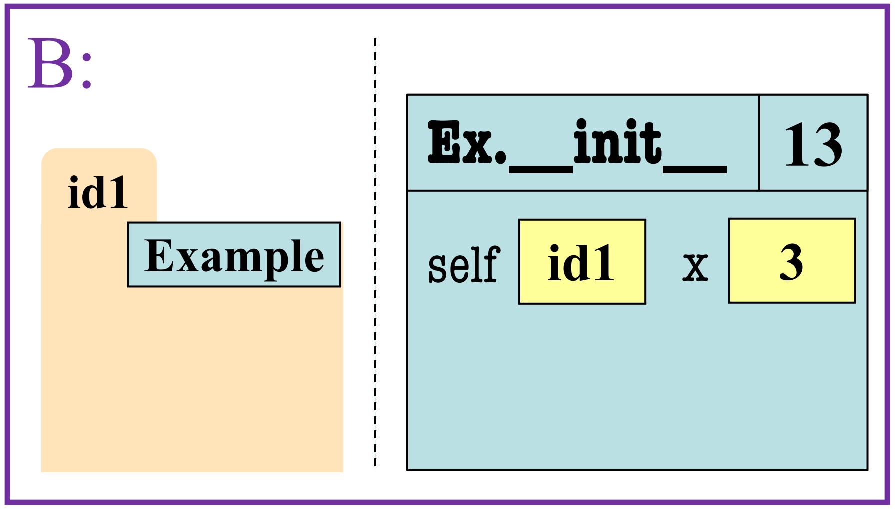
D:



A Class Definition

```
   class Example(object):

12   def __init__(self,x):
13       self.x = x
14
15   def foo(self,y):
16       x = self.bar(y+1)
17       return x
18
19   def bar(self,y):
20       self.x = y-1
21       return self.x
```

```
>>> a = Example(3)
```



What is the **next step**?

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)

```python
class Point3(object):
    """Class for points in 3d space
    Invariant: x is a float
    Invariant y is a float
    Invariant z is a float    """

    def __init__(self,x=0,y=0,z=0):
        """Initializes a new Point3
        Precond: 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):
    """Class for points in 3d space
    Invariant: x is a float
    Invariant y is a float
    Invariant z is a float     """

    def __init__(self,x=0,y=0,z=0):
        """Initializes a new Point3
        Precond: 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):
    """Class for points in 3d space
    Invariant: x is a float
    Invariant y is a float
    Invariant z is a float     """

    def __init__(self,x=0,y=0,z=0):
        """Initializes a p
        P

        lf.z = z
    …
```

Not limited to methods. Can do with any function.