Objects

- Early programming languages did not provide ways to cluster data into coherent collections with well defined interfaces
- Meant that any piece of code to access any part of a data structure
- Lead to occurrence of hard to isolate bugs
- Much better if we can bundle data into packages together with procedures that work on them through well-defined interfaces

Objects

Python supports many different kinds of data:

`1234` **int** `3.14159` **float** `"Hello"` **str**

`[1, 2, 3, 5, 7, 11, 13]` **list**

`{"CA": "California", "MA": "Massachusetts"}` **dict**

Each of the above is an **object.**

Objects have:

- A type (a particular object is said to be an **instance** of a type)
- An internal data representation (primitive or composite)
- A set of procedures for interaction with the object

Example: [1,2,3,4]

- Type: `list`
- Internal data representation
  - int length L, an object array of size S >= L, or
  - A linked list of individual cells

```
<data, pointer to next cell>
```

Example: [1,2,3,4]

- Type: `list`
- Internal data representation
  - int length L, an object array of size S >= L, or
  - A linked list of individual cells

    `<data, pointer to next cell>`

Internal representation is private – users of the objects should not rely on particular details of the implementation. Correct behavior may be compromised if you manipulate internal representation directly.

Example: [1,2,3,4]

- Type: `list`
- Internal data representation
  - int length L, an object array of size S >= L, or
  - A linked list of individual cells

    `<data, pointer to next cell>`
- Procedures for manipulating lists
  - `l[i], l[i:j], l[i,j,k], +, *`
  - `len(), min(), max(), del l[i]`
  - `l.append(…), l.extend(…), l.count(…), l.index(…), l.insert(…), l.pop(…), l.remove(…), l.reverse(…), l.sort(…)`

> Internal representation is private – users of the objects should not rely on particular details of the implementation. Correct behavior may be compromised if you manipulate internal representation directly.

Object-oriented programming (OOP)

- Everything is an **object** and has a **type**
- Objects are a data abstraction that encapsulate
  - Internal representation
  - **Interface** for interacting with object
    - Defines behaviors, hides implementation
    - Attributes: data, methods (procedures)

Object-oriented programming (OOP)

- Everything is an **object** and has a **type**
- Objects are a data abstraction that encapsulate
  - Internal representation
  - **Interface** for interacting with object
    - Defines behaviors, hides implementation
    - Attributes: data, methods (procedures)
- One can
  - Create new instances of objects (explicitly or using literals)
  - Destroy objects
    - Explicitly using `del` or just “forget” about them
    - Python system will reclaim destroyed or inaccessible objects – called “garbage collection”

> Some languages have support for “data hiding” which prevents access to private attributes. Python does not … one is just expected to play by the rules!

Advantages of OOP

- Divide-and-conquer development
  – Implement and test behavior of each class separately
  – Increased modularity reduces complexity
- Classes make it easy to reuse code
  – Many Python modules define new classes
  – Each class has a separate environment (no collision on function names)
  – Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

The power of OOP

- We can bundle together objects that share common attributes with procedures or functions that operate on those attributes
- We can use abstraction to isolate the use of objects from the details of how they are constructed
- We can build layers of object abstractions that inherit behaviors from associated classes of objects
- We can create our own classes of objects on top of Python's basic classes

Defining new types

- In Python, the `class` statement is used to define a new type

```
class Coordinate(object):
```

  ... define attributes here ...

- As with `def`, indentation used to indicate which statements are part of the class definition

Defining new types

- In Python, the `class` statement is used to define a new type

```
class Coordinate(object):
```

… define attributes here …

- As with `def`, indentation used to indicate which statements are part of the class definition
- Classes can inherit attributes from other classes, in this case `Coordinate` inherits from the object classs. `Coordinate` is said to be a **subclass** of `object`, `object` is a **superclass** of `Coordinate`. One can override an inherited attribute with a new definition in the class statement.

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Method is another name for a procedural attribute, or a procedure that "belongs" to this class

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the object as the first argument. By convention, we use `self` as the name of the first argument of methods.

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the object as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The "." operator is used to access an attribute of an object. So the `__init__` method above is defining two attributes for the new `Coordinate` object: `x` and `y`.

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the object as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The "." operator is used to access an attribute of an object. So the `__init__` method above is defining two attributes for the new `Coordinate` object: `x` and `y`.

When accessing an attribute of an instance, start by looking within the class definition, then move up to the definition of a superclass, then move to the global environment

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the object as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The "." operator is used to access an attribute of an object. So the `__init__` method above is defining two attributes for the new `Coordinate` object: `x` and `y`.

Data attributes of an instance are often call **instance variables.**

Creating an instance

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

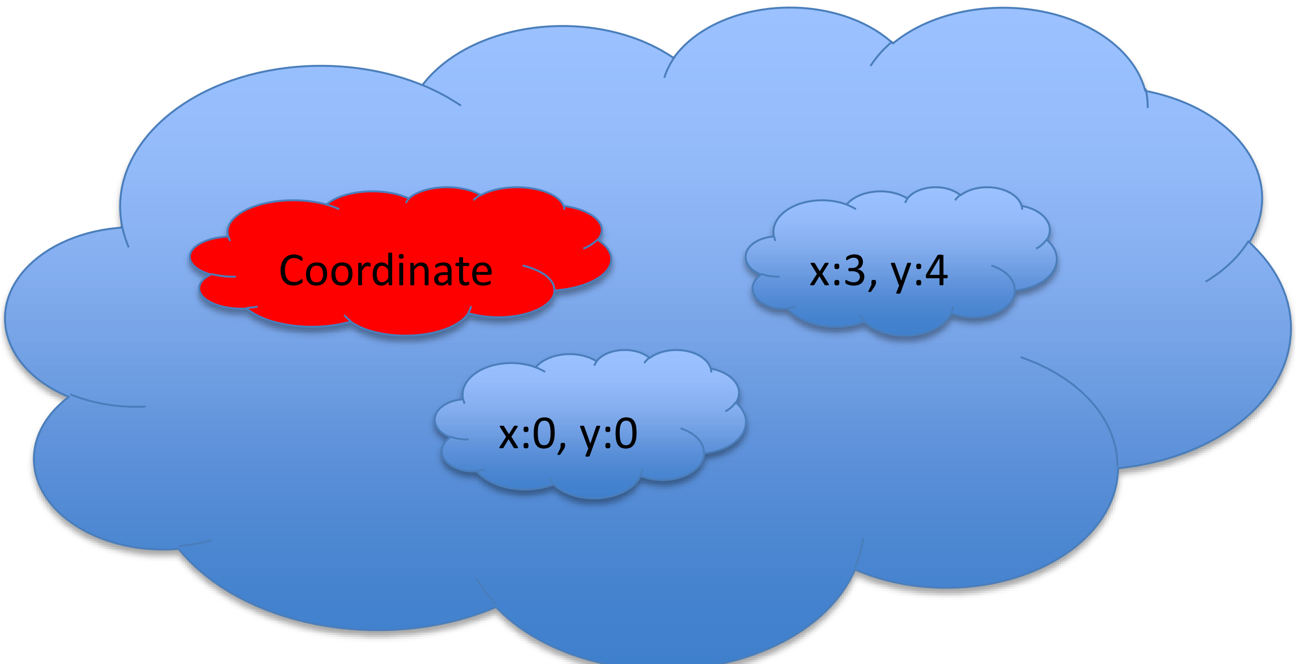
```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

c = Coordinate(3,4)
origin = Coordinate(0,0)
print c.x, origin.x
```

The expression
`classname(values…)`
creates a new object of type classname and then calls its `__init__` method with the new object and `values…` as the arguments. When the method is finished executing, Python returns the initialized object as the value.

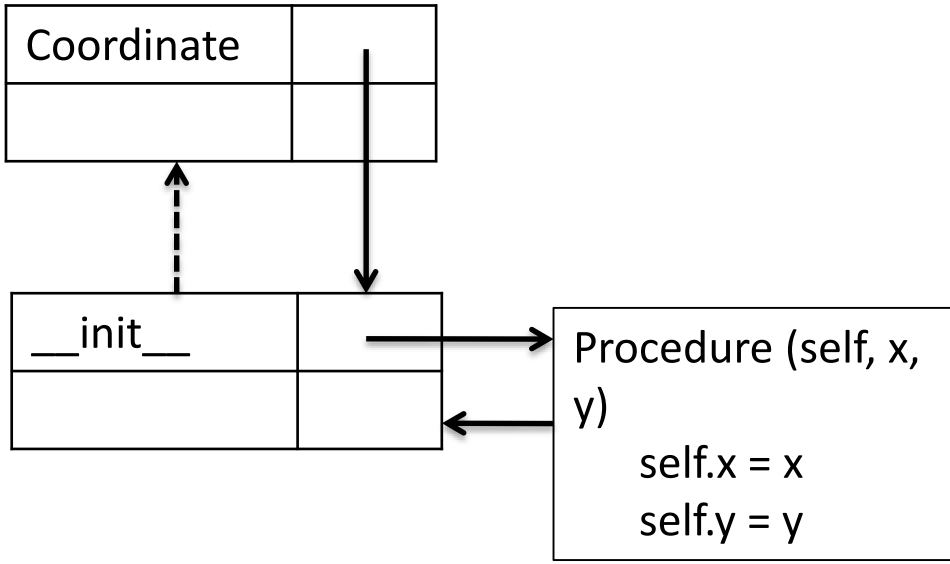
Note that don't provide argument for self, Python does this automatically

Visualizing this idea



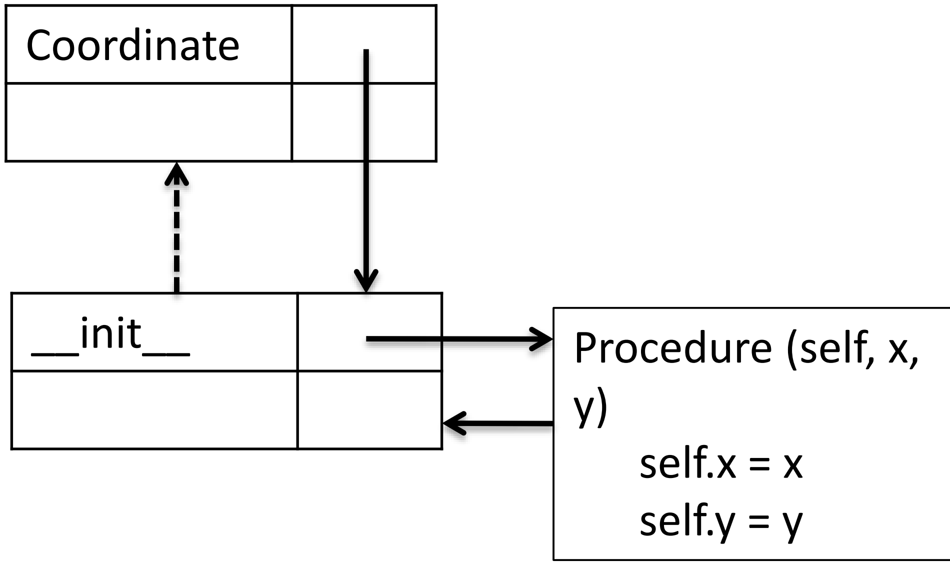
An environment view of classes

- Class definition creates a binding of class name in global environment to a new frame or environment
- That frame contains any attribute bindings, either variables or local procedures
- That frame also knows the parent environment from which it can inherit



An environment view of classes

- In this case, the only attribute is a binding of a name to a procedure
- But if a class definition bound local variables as part of its definition, those would also be bound in this new environment

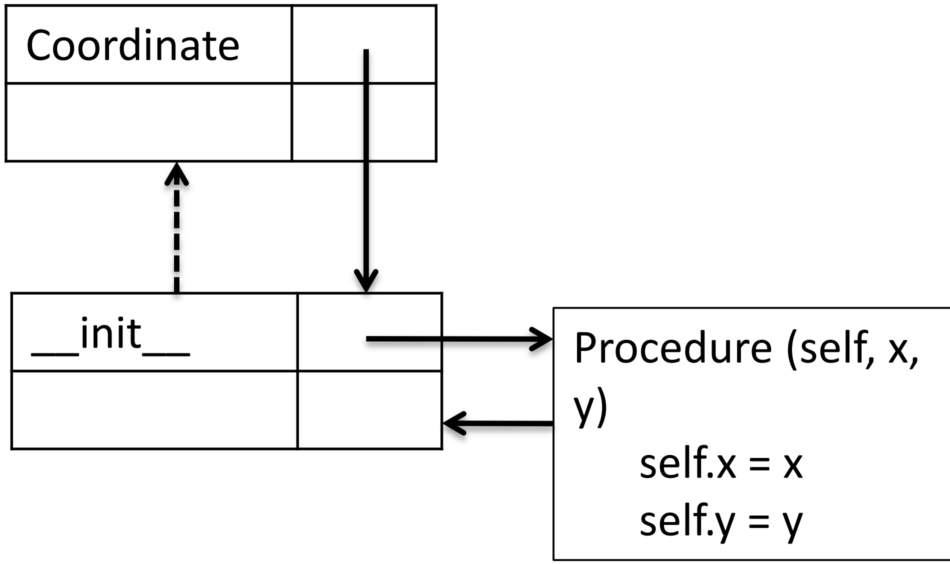


An environment view of classes

- We can access parts of a class using

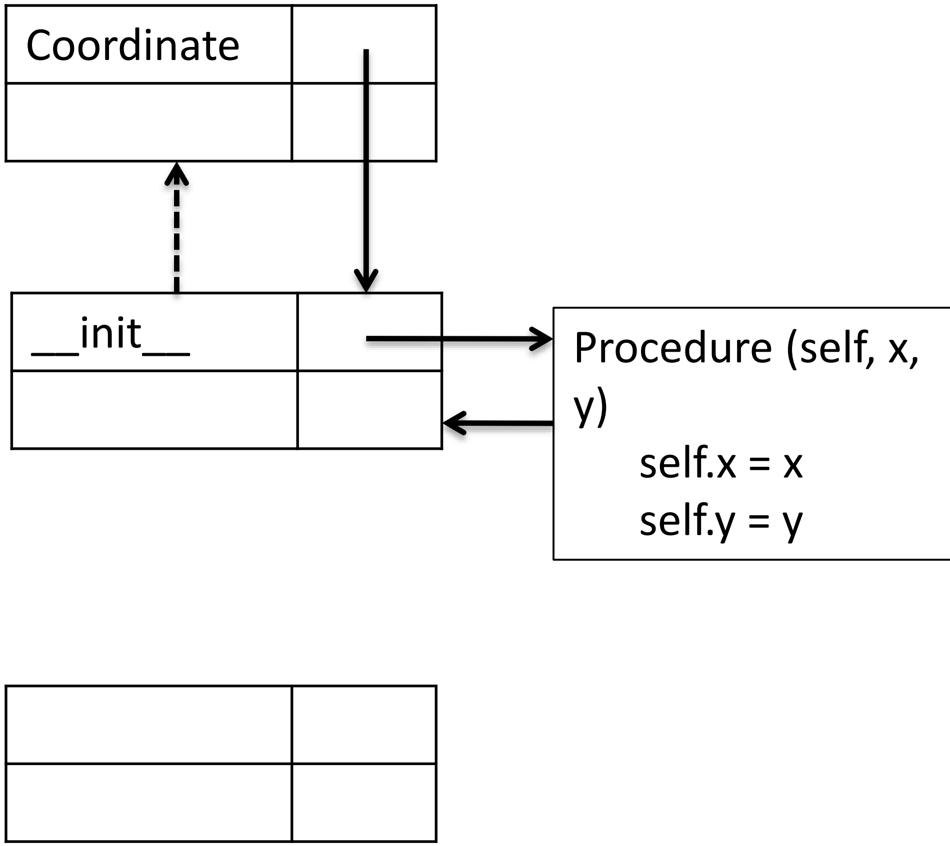
`Coordinate.__init__`

- Python interprets this by finding the binding for the first expression (which is a frame), and then using the standard rules to lookup the value for the next part of the expression in that frame



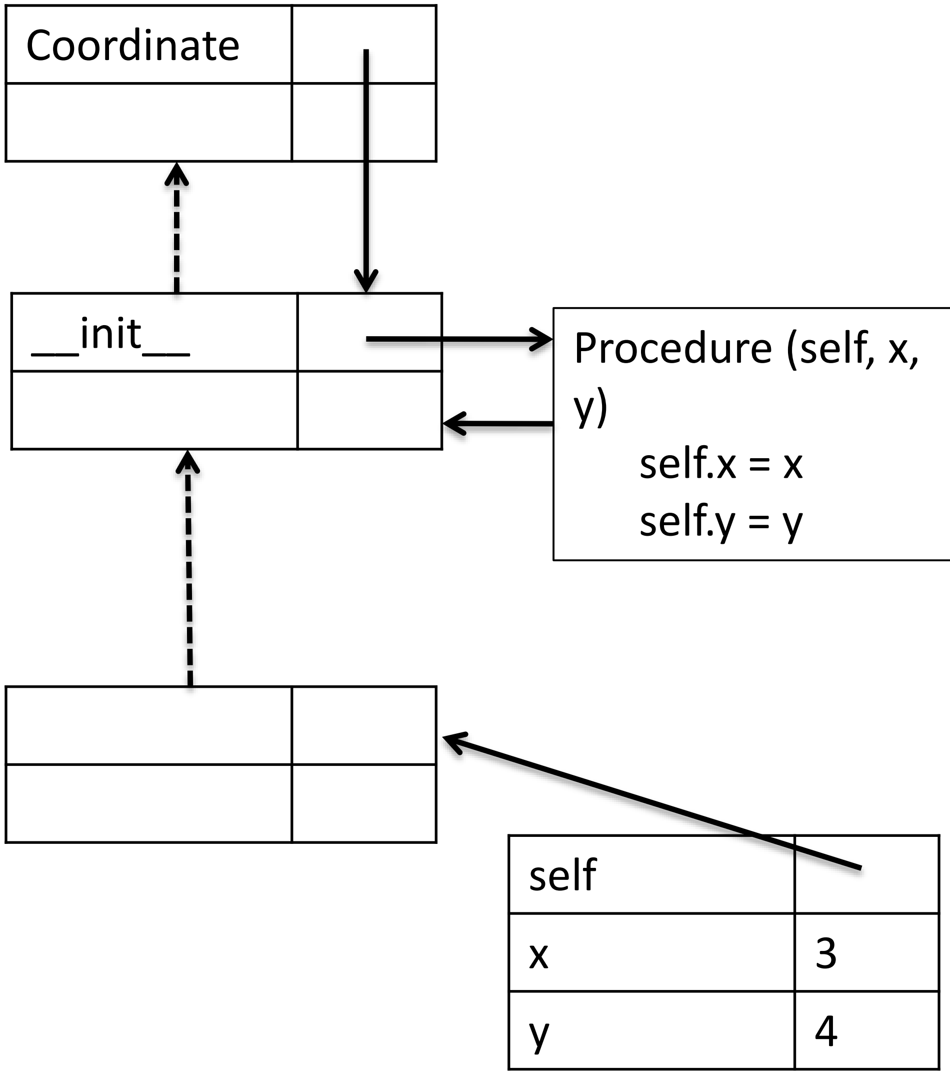
An environment view of classes

- Suppose the class is invoked
  - c = Coordinate(3,4)
- A new frame is created (this is the instance)



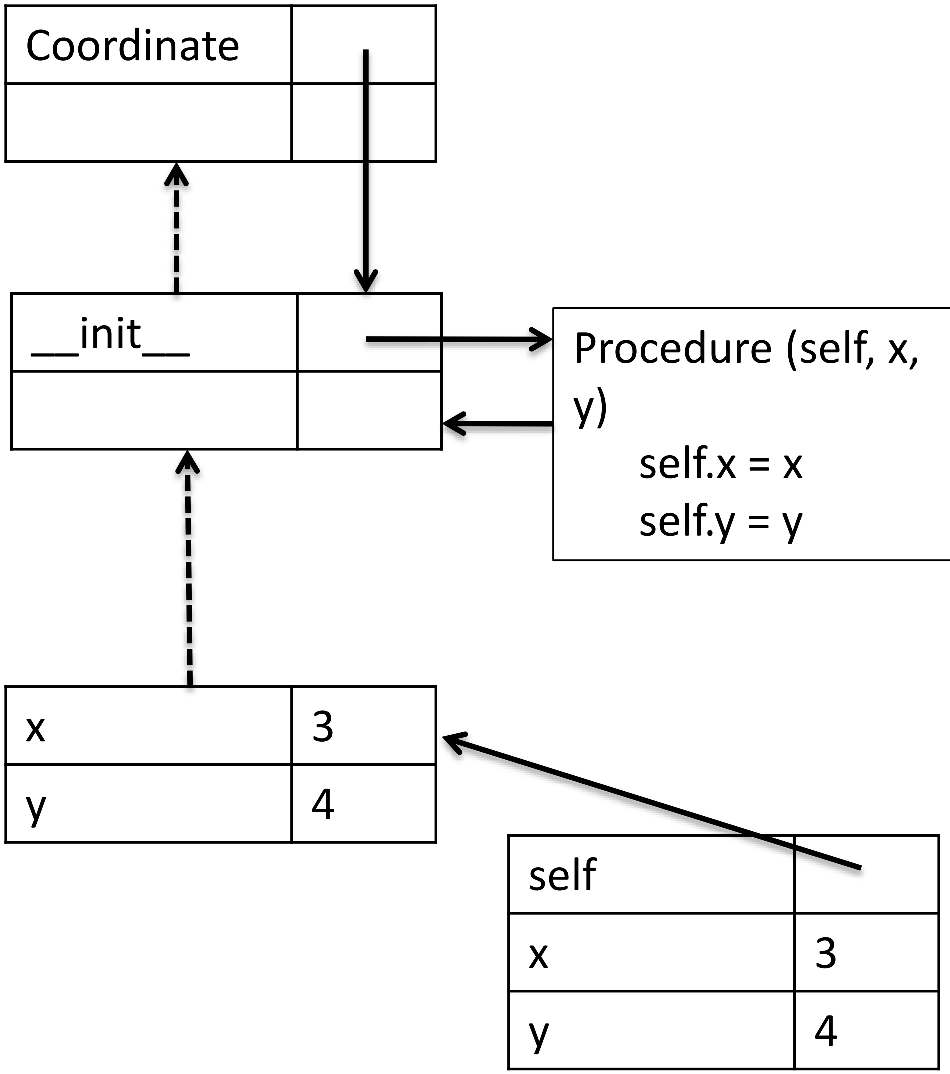
An environment view of classes

- Suppose the class is invoked
  - c = Coordinate(3,4)
- A new frame is created (this is the instance)
- The __init__ method is then called, with self bound to this object, plus any other arguments
- The instance knows about the frame in which __init__ was called



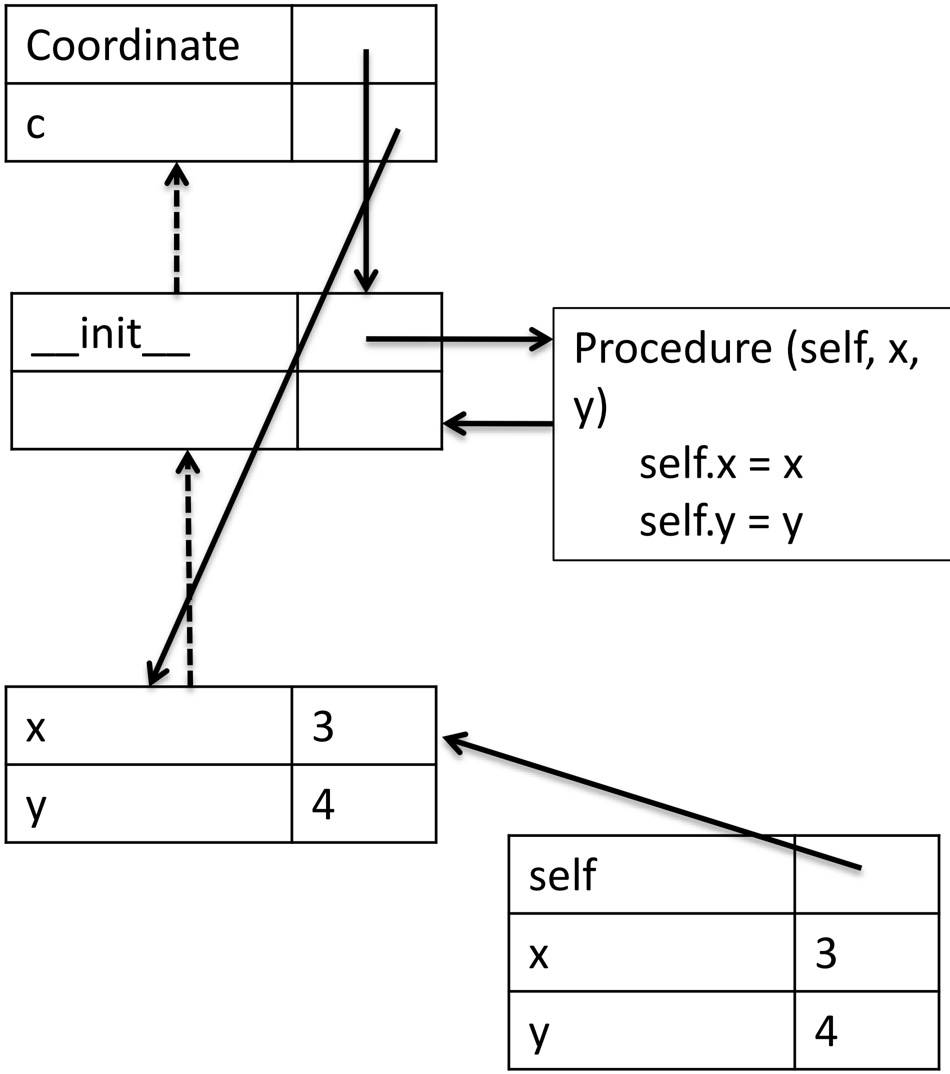
An environment view of classes

- Suppose the class is invoked
  - c = Coordinate(3,4)
- A new frame is created (this is the instance)
- The __init__ method is then called, with self bound to this object, plus any other arguments
- Evaluating the body of __init__ creates bindings in the frame of the instance



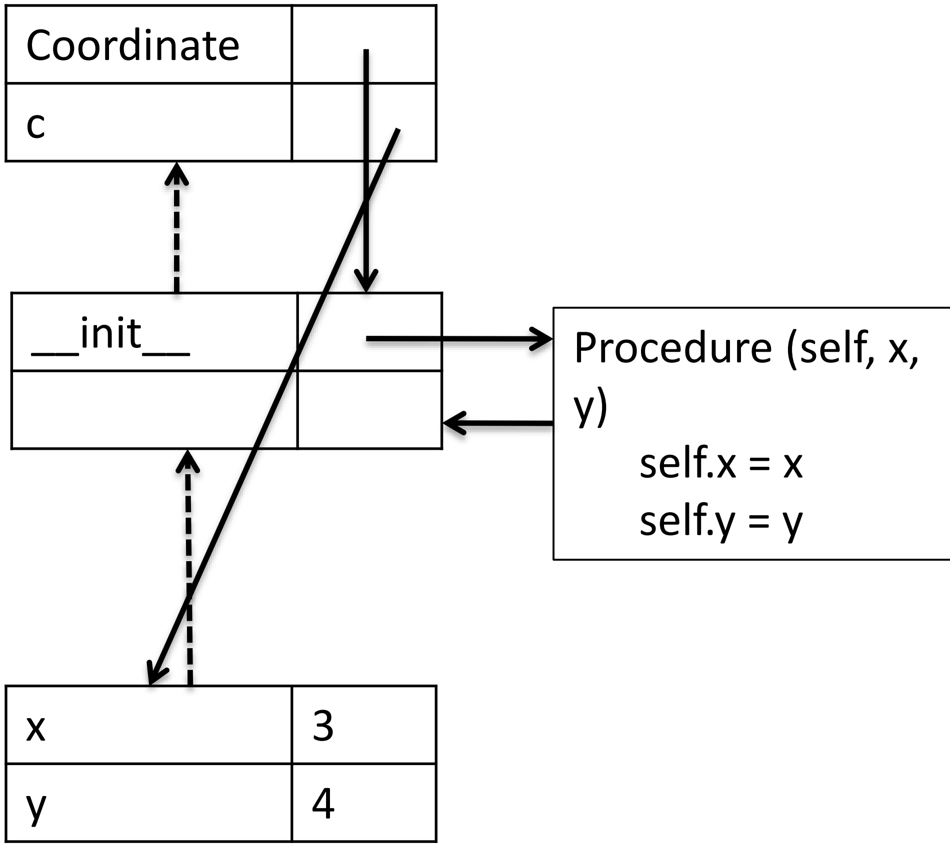
An environment view of classes

- Suppose the class is invoked
  - c = Coordinate(3,4)
- A new frame is created (this is the instance)
- The __init__ method is then called, with self bound to this object, plus any other arguments
- Evaluating the body of __init__ creates bindings
- Finally the frame created by the class call is returned, and bound in the global environment



An environment view of classes

- Given such bindings, calls to attributes are easily found
- c.x will return 3 because c points to a frame, and within that frame x is locally bound

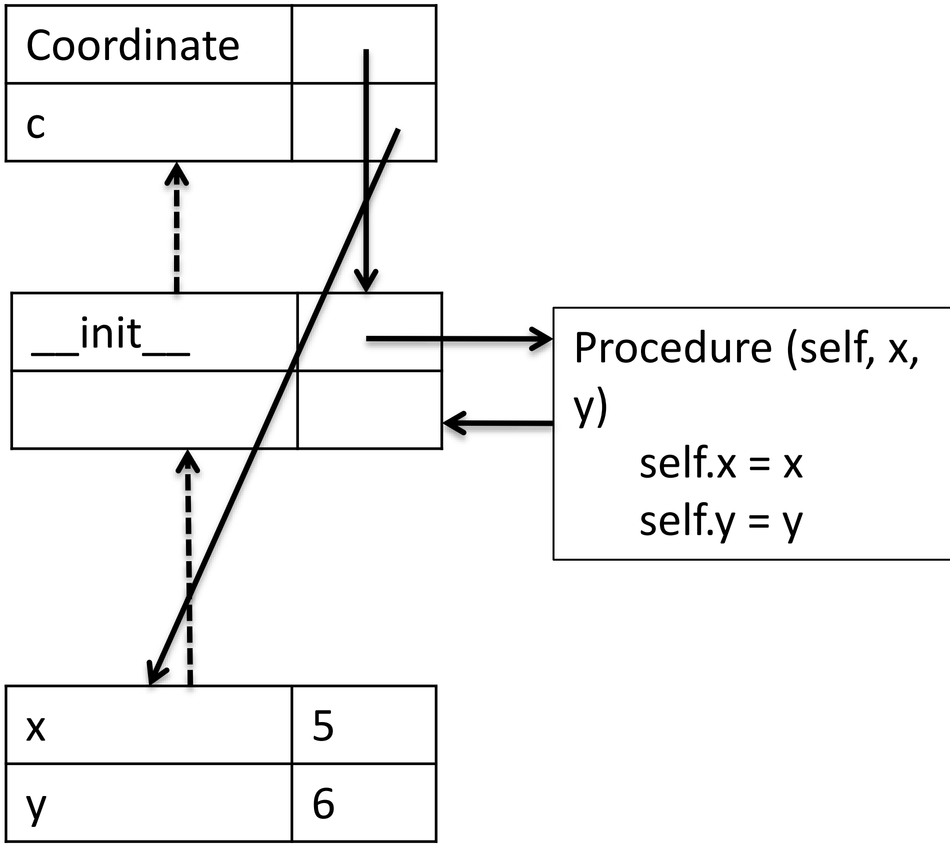


An environment view of classes

- Given such bindings, calls to attributes are easily found
- c.x will return 3 because c points to a frame, and within that frame x is locally bound
- Note that c has access to any binding in the chain of environments

```
c.__init__(5,6)
```

- will change the bindings for x and y within c

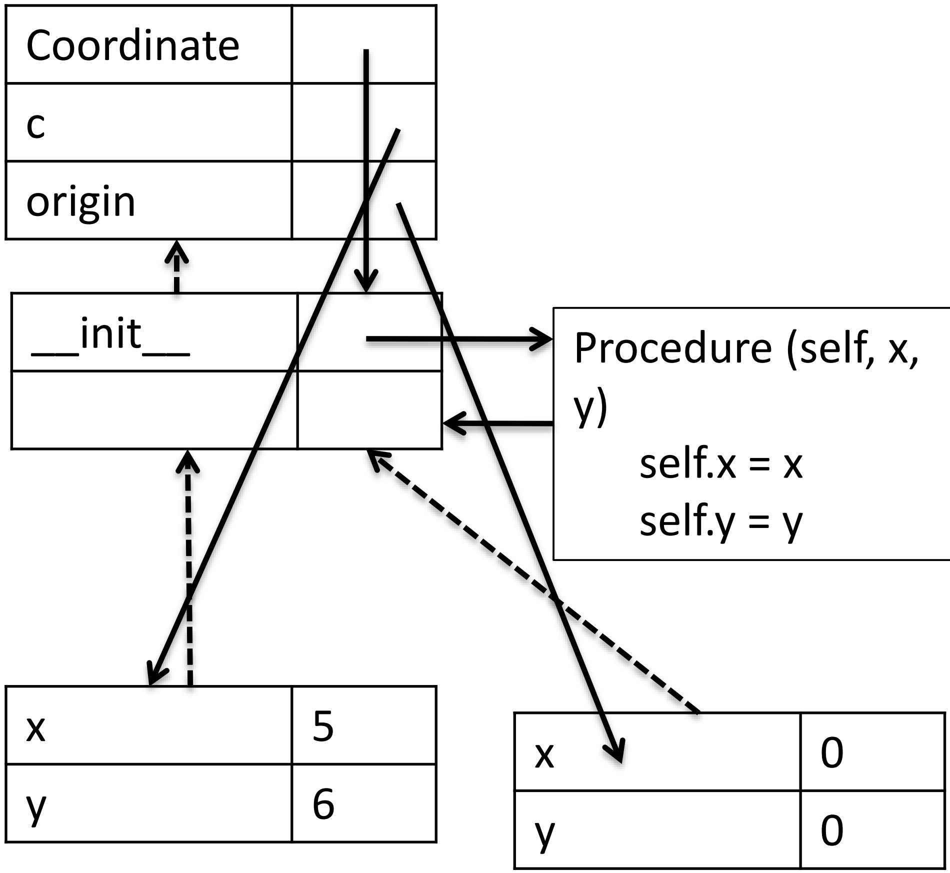


An environment view of classes

- Given such bindings, calls to attributes are easily found
- c.x will return 3 because c points to a frame, and within that frame x is locally bound
- Creating a new instance, creates a new environment, e.g.

```
Origin = Coordinate(0,0)
```

- This shares information within the class environment



Print representation of an object

- Left to its own devices, Python uses a unique but uninformative print presentation for an object

```
>>> print c
<__main__.Coordinate object at 0x7fa918510488>
```

Print representation of an object

- Left to its own devices, Python uses a unique but uninformative print presentation for an object

```
>>> print c
<__main__.Coordinate object at 0x7fa918510488>
```

- One can define a `__str__` method for a class, which Python will call when it needs a string to print. This method will be called with the object as the first argument and should return a `str`.

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "<"+self.x+","+self.y+">"
```

Print representation of an object

- Left to its own devices, Python uses a unique but uninformative print presentation for an object

```
>>> print c
<__main__.Coordinate object at 0x7fa918510488>
```

- One can define a `__str__` method for a class, which Python will call when it needs a string to print. This method will be called with the object as the first argument and should return a `str`.

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "<"+self.x+","+self.y+">"
>>> print c
<3,4>
```

Type of an Object

- We can ask for the type of an object

```
>>> print type(c)
<class __main__.Coordinate>
```

- This makes sense since

```
>>> print Coordinate, type(Coordinate)
<class __main__.Coordinate> <type ‘type’>
```

Type of an Object

- We can ask for the type of an object

```
>>> print type(c)
<class __main__.Coordinate>
```

- This makes sense since

```
>>> print Coordinate, type(Coordinate)
<class __main__.Coordinate> <type 'type'>
```

- Use `isinstance()` to check if an object is a Coordinate

```
>>> print isinstance(c, Coordinate)
True
```

Adding other methods

- Can add our own methods, not just change built-in ones

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "<"+self.x+","+self.y+">"
    def distance(self,other):
        return math.sqrt(sq(self.x - other.x)
                         + sq(self.y -
other.y))
```

Example: a set of integers

- Create a new type to represent a set (or collection) of integers
  - Initially the set is empty
  - A particular integer appears only once in a set
    - This constraint, called a **representational invariant**, is enforced by the code in the methods.
- Internal data representation
  - Use a list to remember the elements of a set
- Interface
  - `insert(e)` – insert integer `e` into set if not there
  - `member(e)` – return `True` if integer `e` is in set, `False` else
  - `remove(e)` – remove integer `e` from set, error if not present

An implementation

```
class intSet(object):
    """An intSet is a set of integers
    The value is represented by a list of ints, self.vals.
    Each int in the set occurs in self.vals exactly once."""

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        return '{' + ','.join([str(e) for e in self.vals]) + '}'

  # other procedural attributes
```

An implementation

```
class intSet(object):
    """An intSet is a set of integers
    The value is represented by a list of ints, self.vals.
    Each int in the set occurs in self.vals exactly once."""

  # other procedural attributes

     def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

     def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals
```

An implementation

```
class intSet(object):
    """An intSet is a set of integers
    The value is represented by a list of ints, self.vals.
    Each int in the set occurs in self.vals exactly once."""

  # other procedural attributes

     def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)


    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')
```