Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 17
Variables and Scoping

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

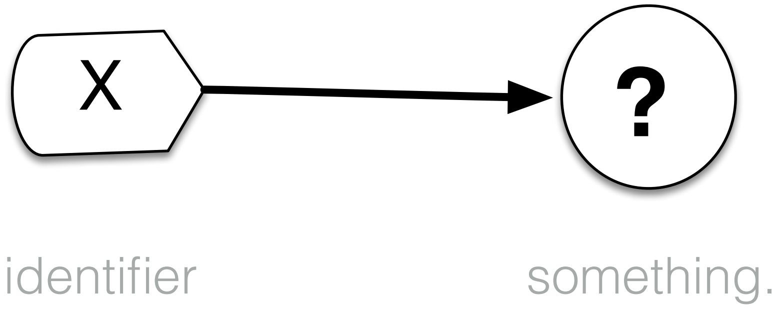
Objectives

- To revise how **variables and values** are represented internally in **Python**
- To understand names and scopes.

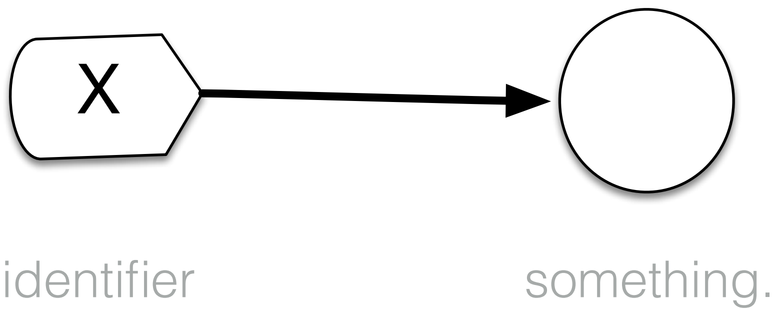
Variable representation

- What is a **variable**?
  *A name (**identifier**) of “something”*

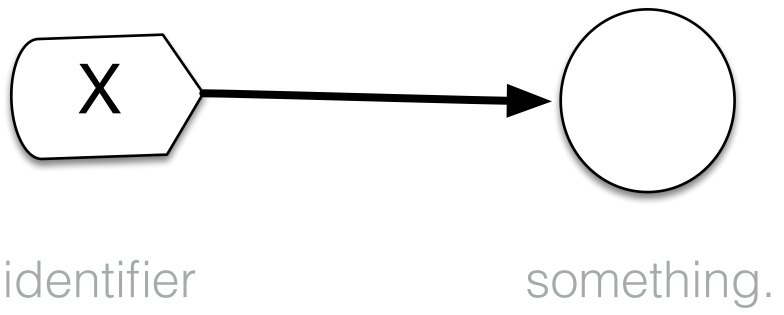
- The name (in almost all languages) <u>refers to a memory address.</u> That memory address contains…“something



Variable representation

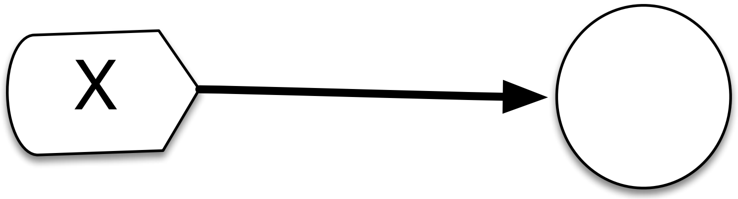


Variable representation



- The content depends … on the language!

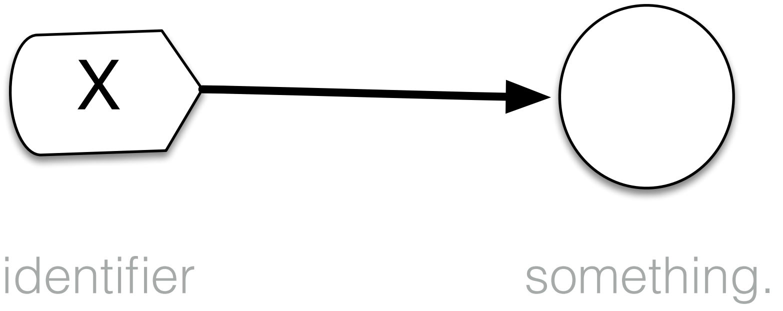
Variable representation



identifier something.

- The content depends … on the language!
- In Python: it is **a label** reference to the memory location containing
    - The **data**
    - The **type of the data**
    - Other stuff…

Variable representation



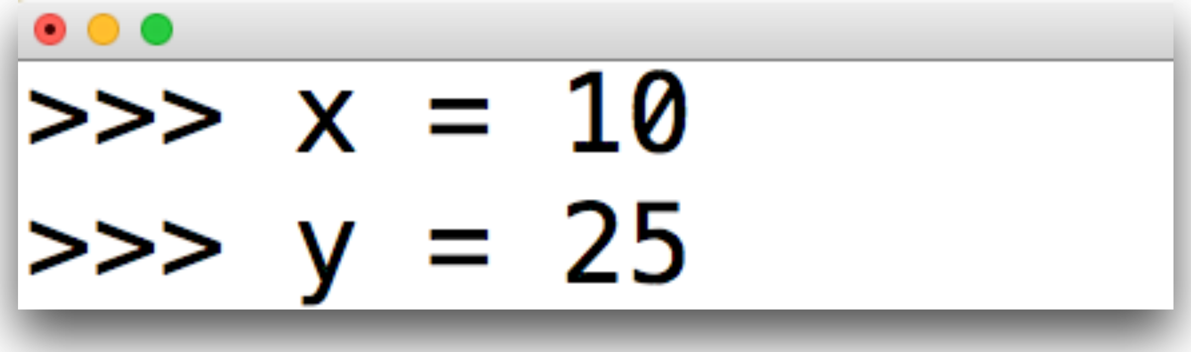
- The content depends … on the language!

- In Python: it is **a label** reference to the memory location containing

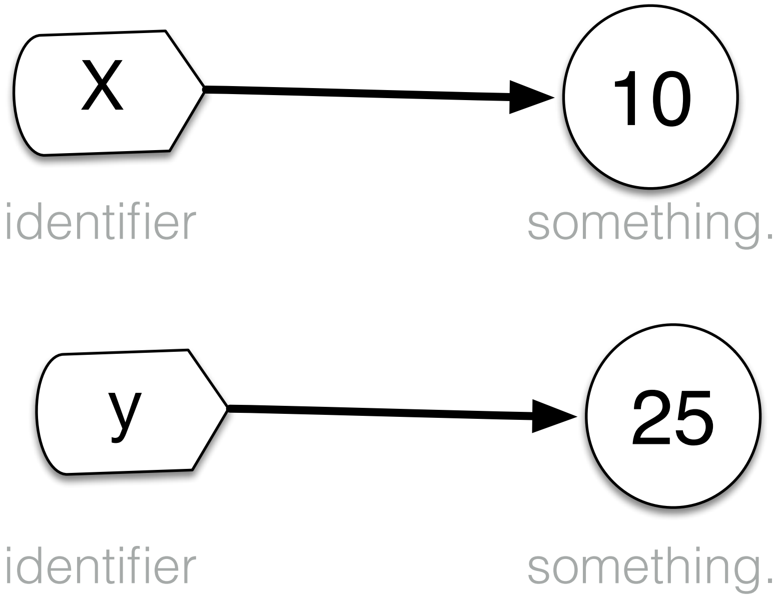
  - The **data**
  - The **type of the data**
  - Other stuff…

The “object”

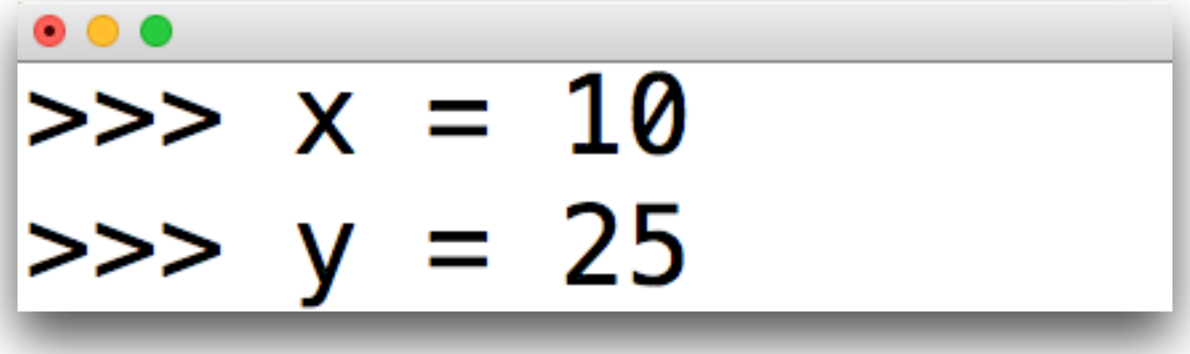
Variable representation in Python



```
>>> x = 10
>>> y = 25
```

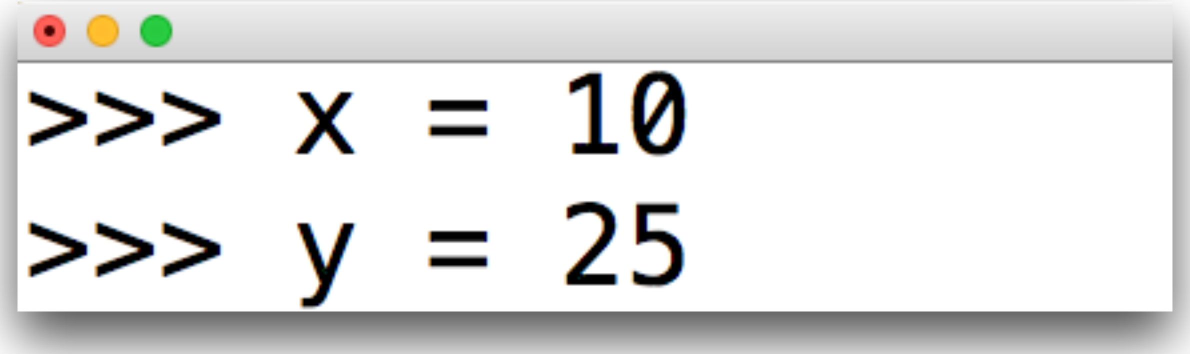


Variable representation in Python

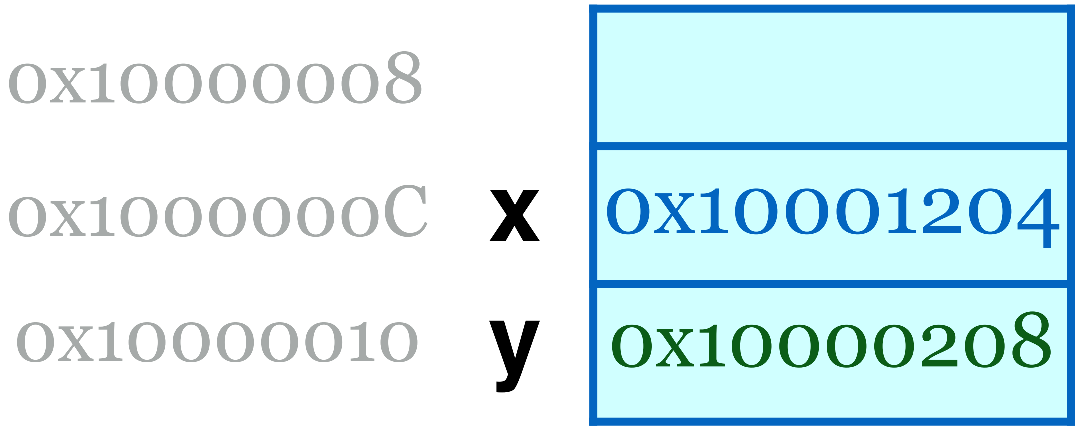


```
>>> x = 10
>>> y = 25
```

Variable representation in Python

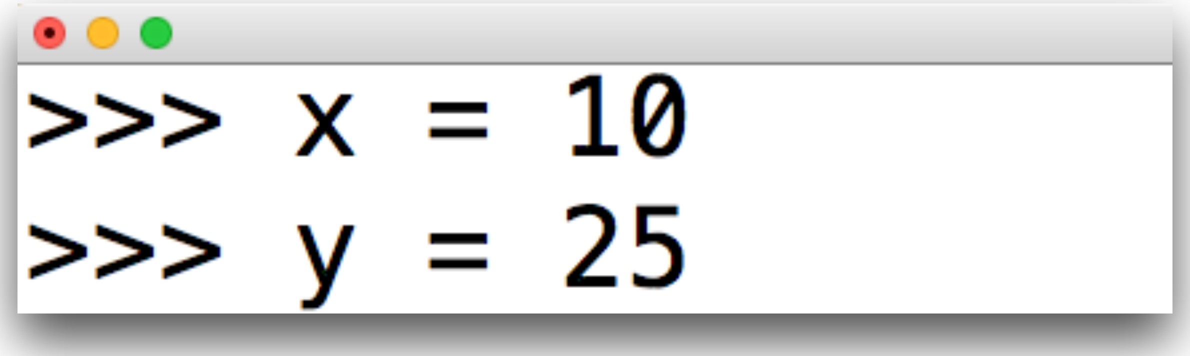


```
>>> x = 10
>>> y = 25
```

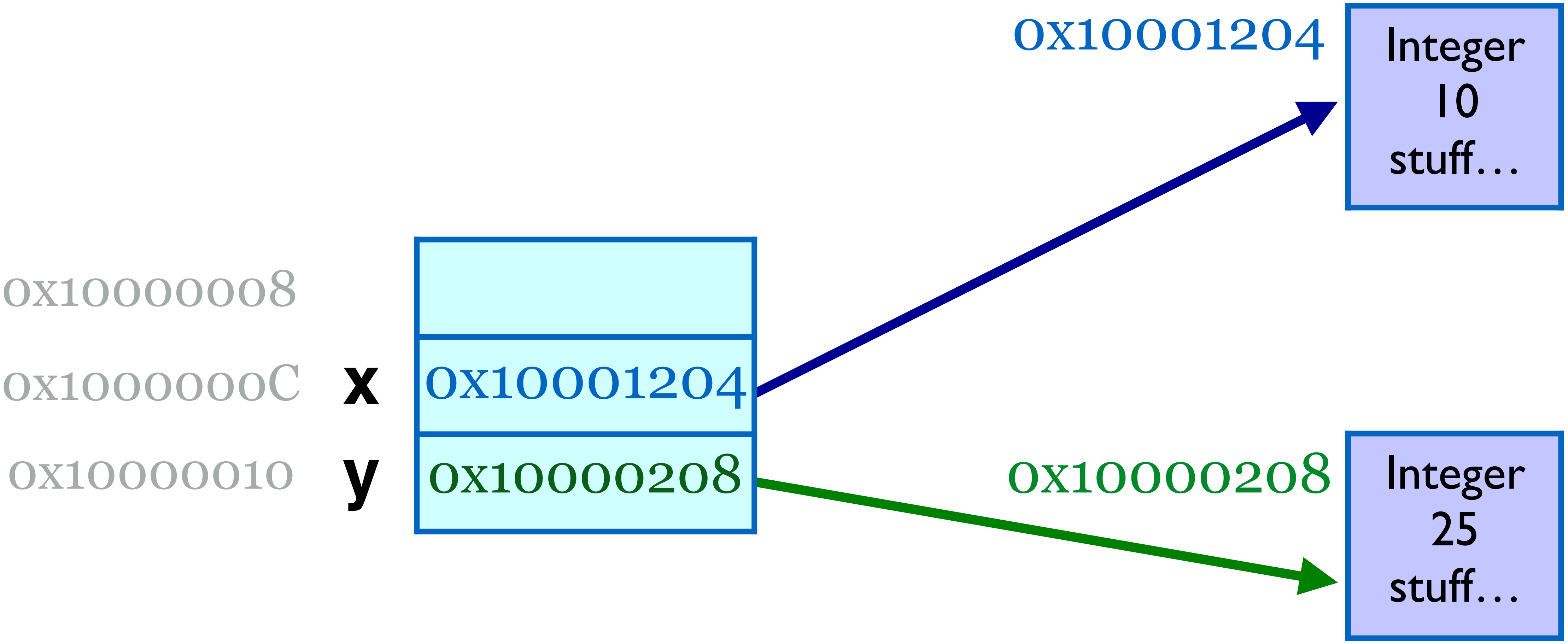


0x10000008		
0x1000000C	x	0x10001204
0x10000010	y	0x10000208

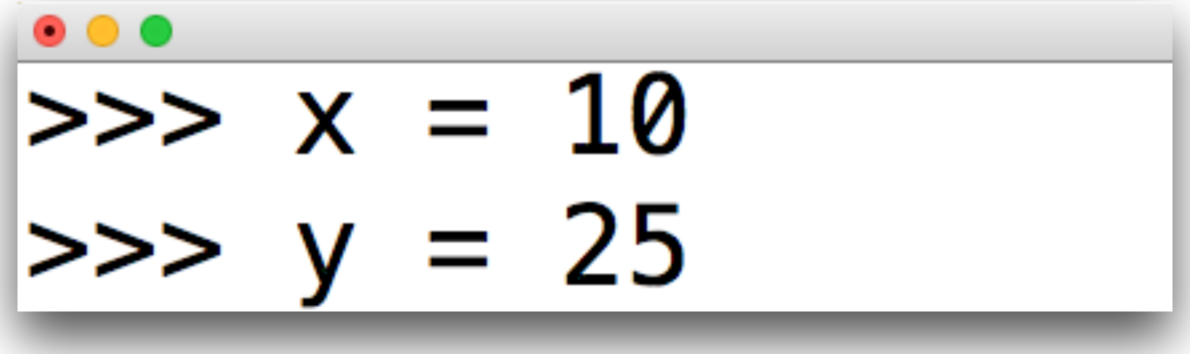
Variable representation in Python



```
>>> x = 10
>>> y = 25
```



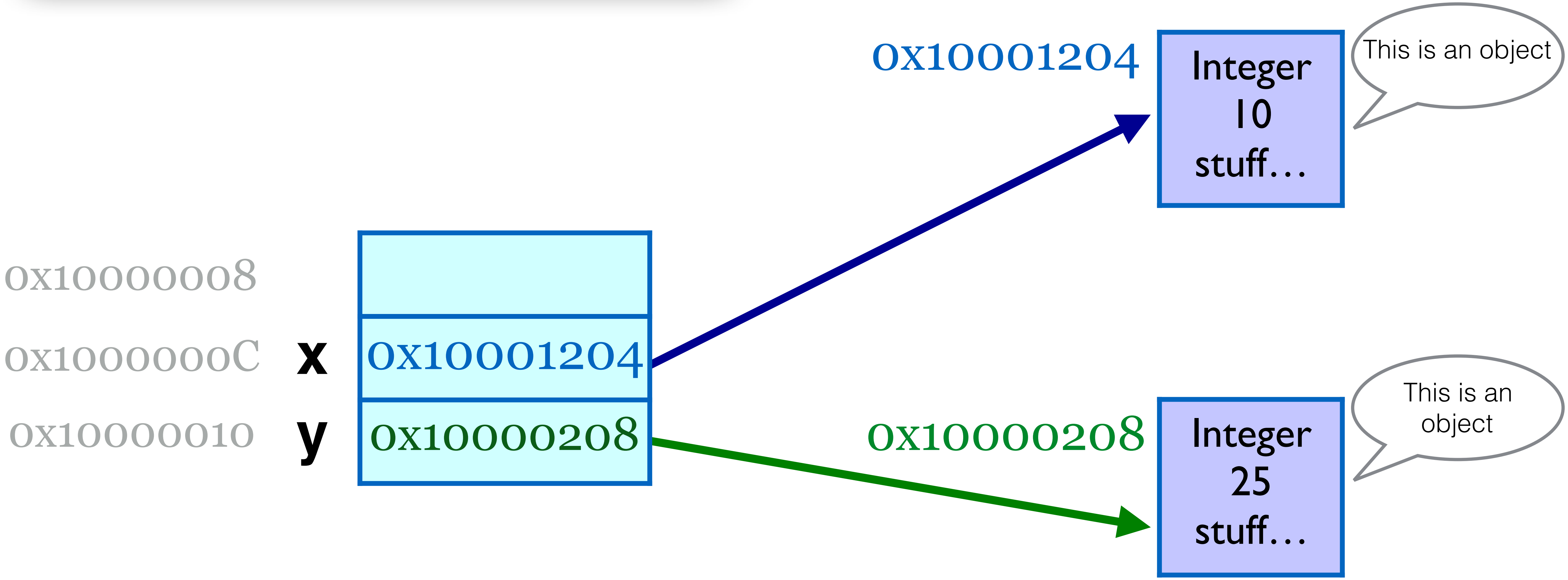
Variable representation in Python



```
>>> x = 10
>>> y = 25
```

- The **data**
- The **type of the data**
- Other stuff…

The “object”

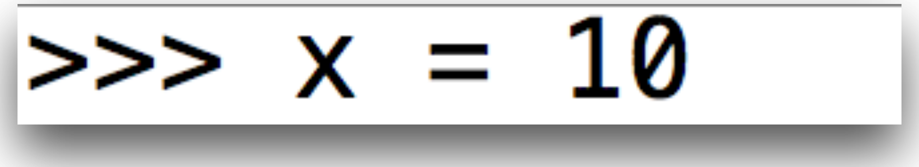


Creating variables in Python

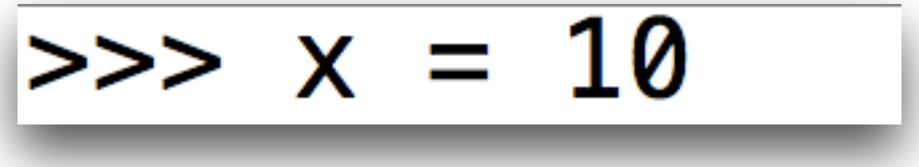
- A variable is **created** when you first **assign** it a **value**

Creating variables in Python

- A variable is **created** when you first **assign** it a **value**
- In many other languages, variables can be created without a value (“declared”)

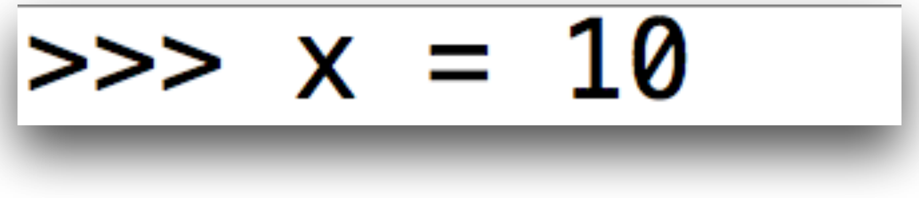


```
>>> x = 10
```



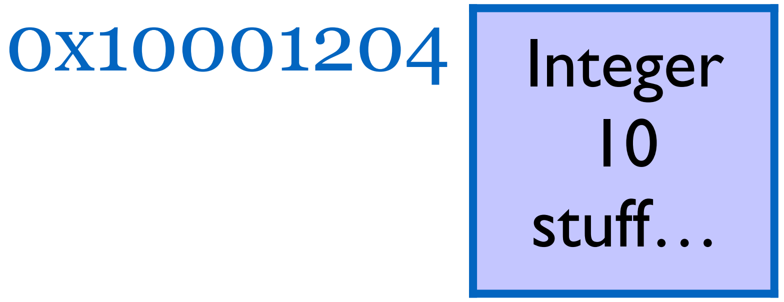
```
>>> x = 10
```

1. **Creates an object** to represent 10, starting at some address



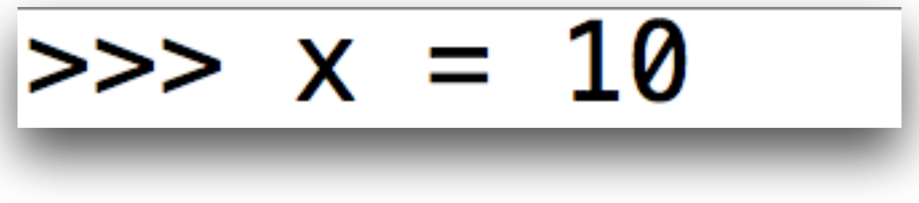
```
>>> x = 10
```

1. **Creates an object** to represent 10, starting at some address



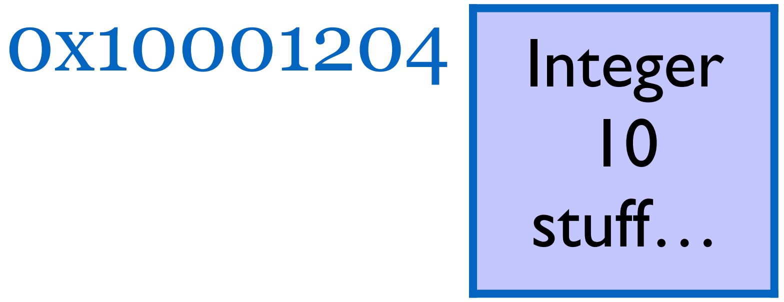
0x10001204

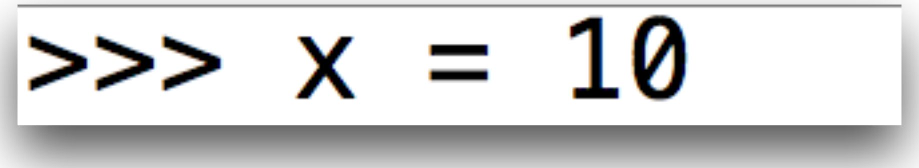
Integer
10
stuff…



```
>>> x = 10
```

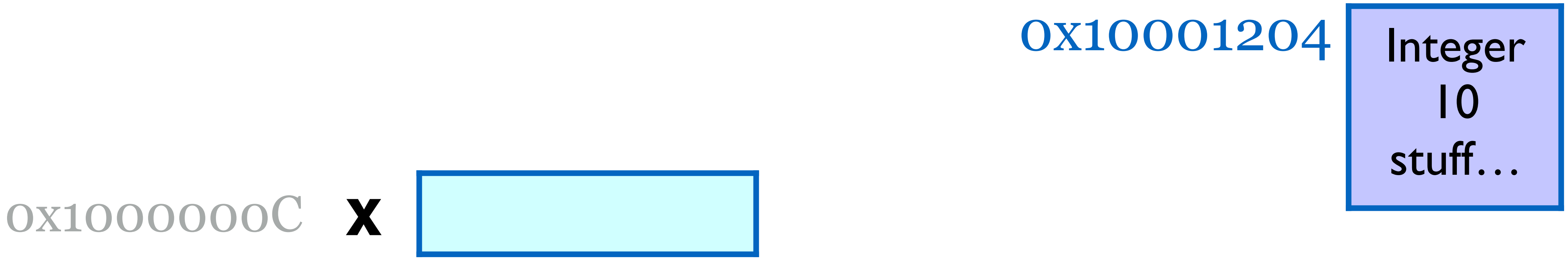
1. **Creates an object** to represent 10, starting at some address
2. Creates the variable **x** if it does not exist

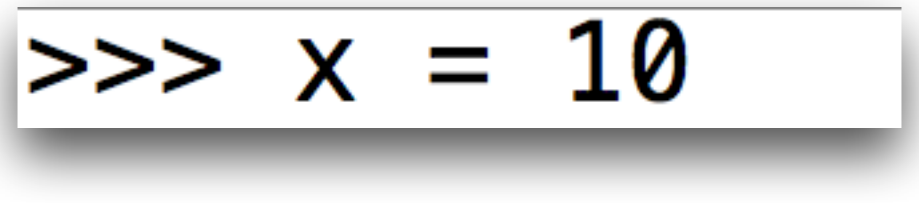




```
>>> x = 10
```

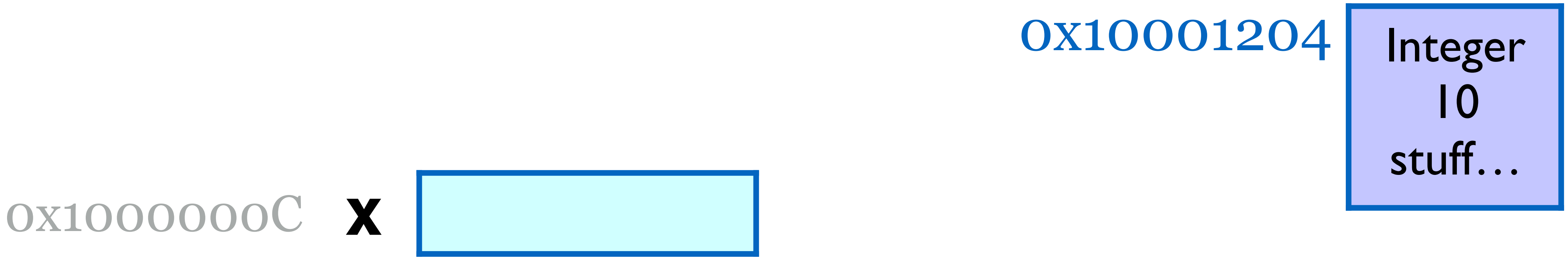
1. **Creates an object** to represent 10, starting at some address
2. Creates the variable **x** if it does not exist





```
>>> x = 10
```

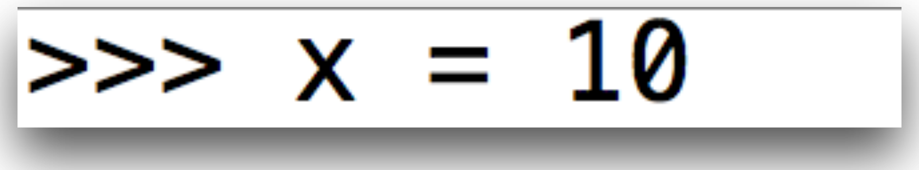
1. **Creates an object** to represent 10, starting at some address
2. Creates the variable **x** if it does not exist
3. **Links it** with the object created (assigns the address to **x**)



0x10001204

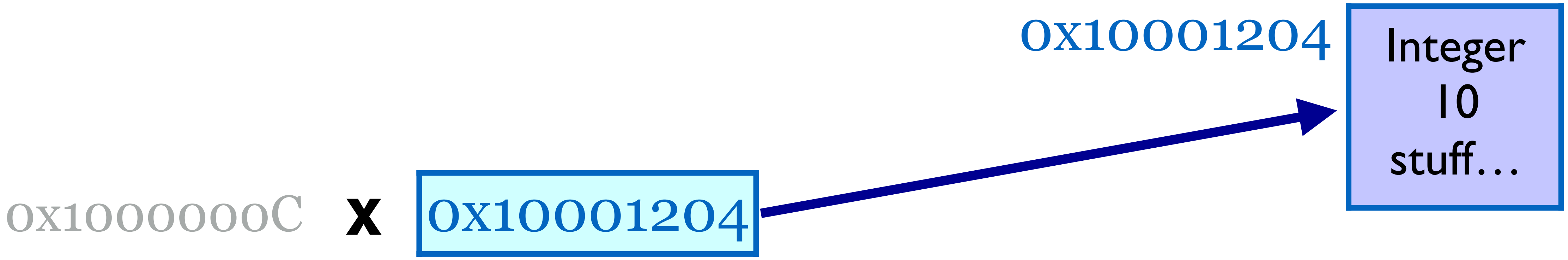
Integer
10
stuff…

0x1000000C **x**



```
>>> x = 10
```

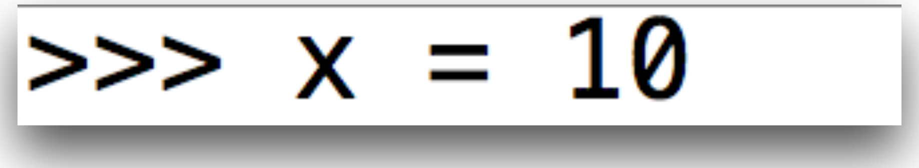
1. **Creates an object** to represent 10, starting at some address
2. Creates the variable **x** if it does not exist
3. **Links it** with the object created (assigns the address to **x**)



0x10001204

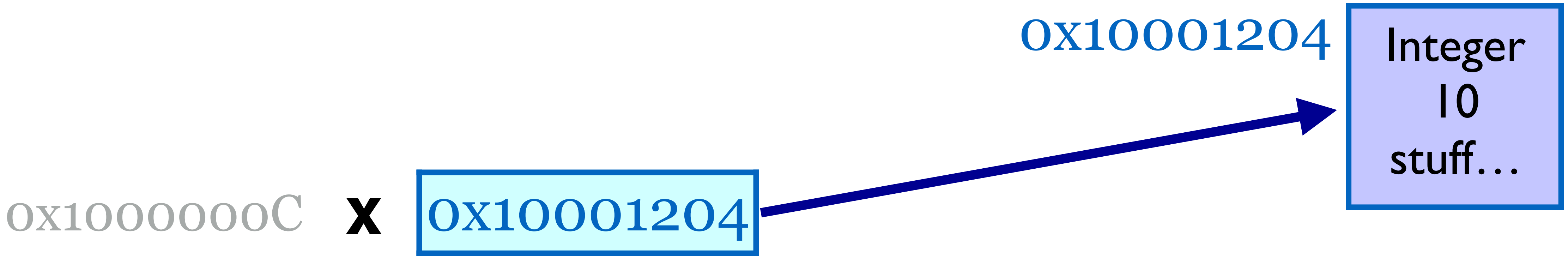
Integer
10
stuff…

0x1000000C **x** 0x10001204

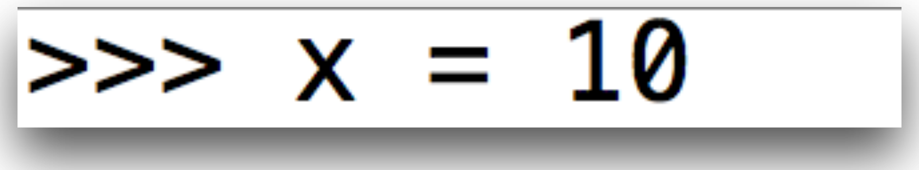


```
>>> x = 10
```

1. **Creates an object** to represent 10, starting at some address
2. Creates <u>the variable</u> **x** if it does not exist
3. **Links it** with the object created (assigns the address to **x**)

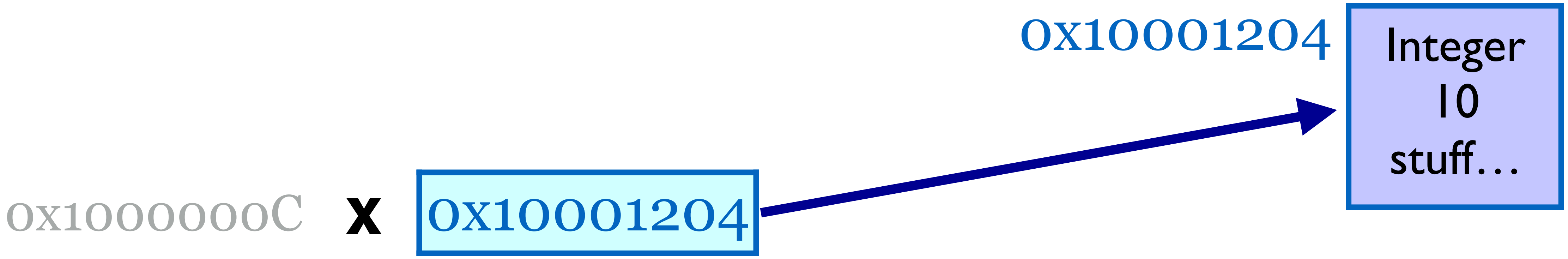


Consequence:
Variables do not have a type. Types are associated with values (i.e., with object)



```
>>> x = 10
```

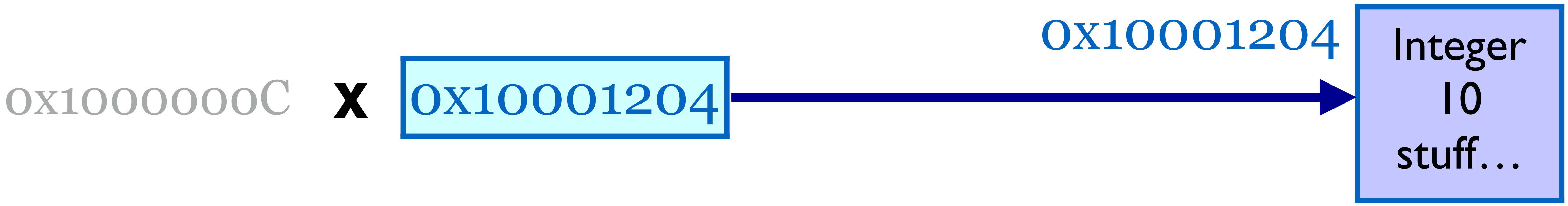
1. **Creates an object** to represent 10, starting at some address
2. Creates <u>the variable</u> **x** if it does not exist
3. **Links it** with the object created (assigns the address to **x**)



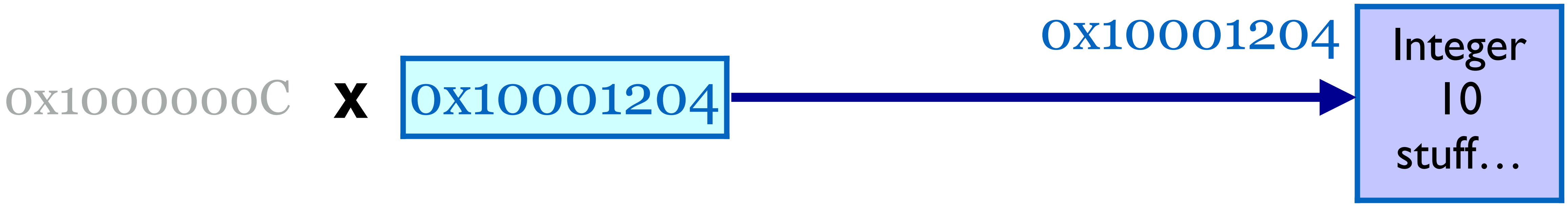
Consequence:
Variables do not have a type. Types are associated with values (i.e., with object)

You can assign values of different types to the same variable

Our visualisation of objects in Python

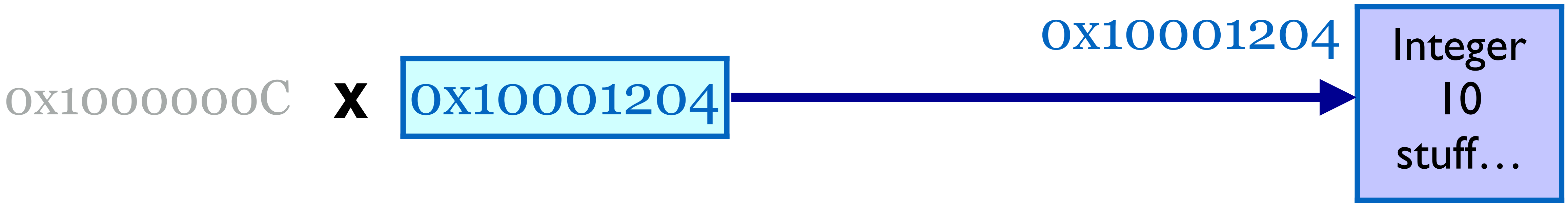


Our visualisation of objects in Python

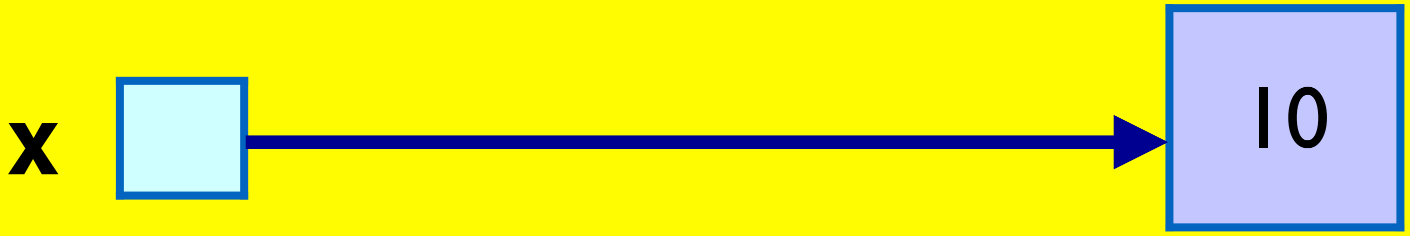


- We will only display values within the object
- Ignore the exact value of the references (i.e., the address)

Our visualisation of objects in Python



- We will only display values within the object
- Ignore the exact value of the references (i.e., the address)



And once variables are created?

- Variables are **always** labels to where in the memory the objects are stored.

And once variables are created?

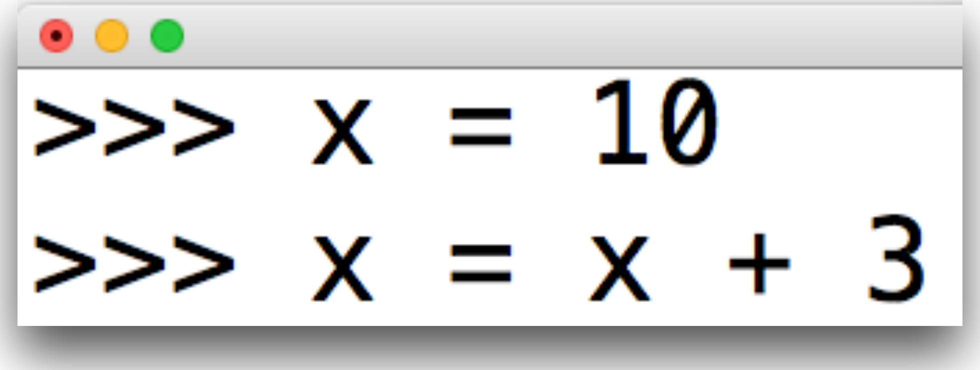
- Variables are **always** labels to where in the memory the objects are stored.
- Assignments do not alter the object itself. They only alter the reference.

And once variables are created?

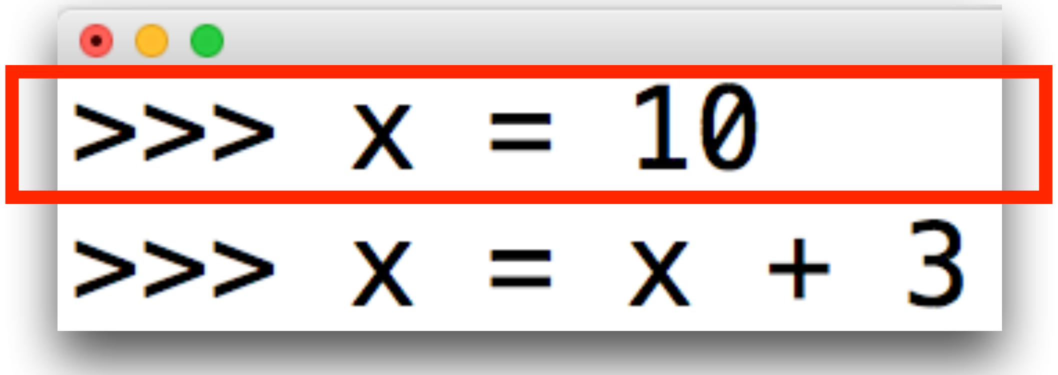
- Variables are **always** labels to where in the memory the objects are stored.

- Assignments do not alter the object itself. They only alter the reference.

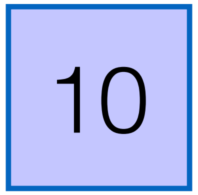
- The variable will refer to a different object.



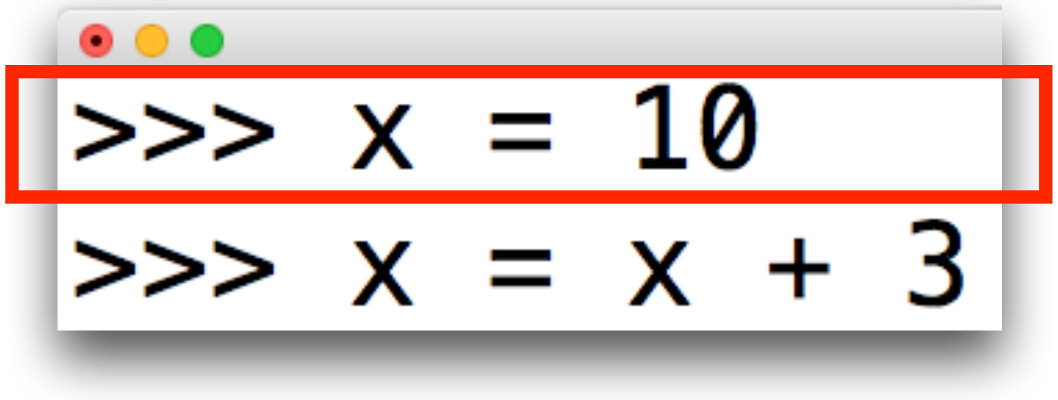
```
>>> x = 10
>>> x = x + 3
```



```
>>> x = 10
>>> x = x + 3
```

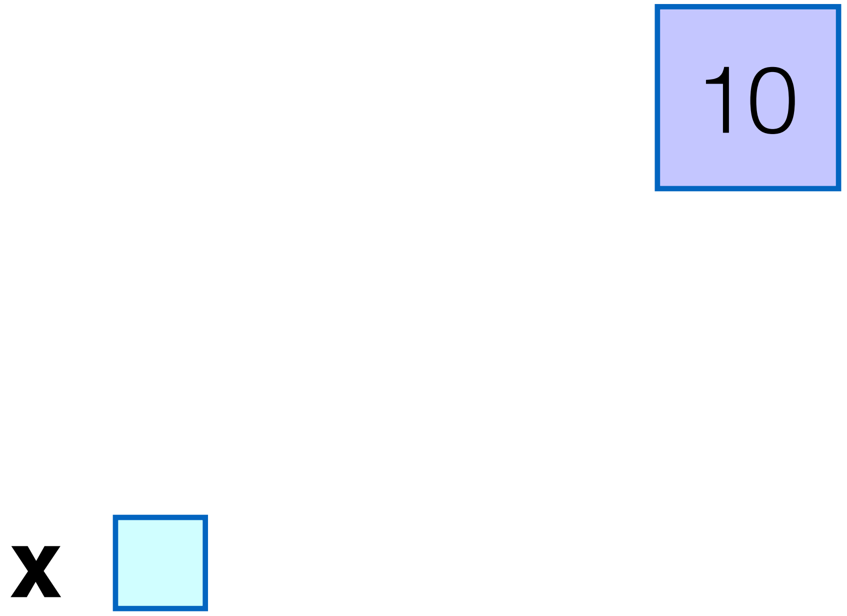


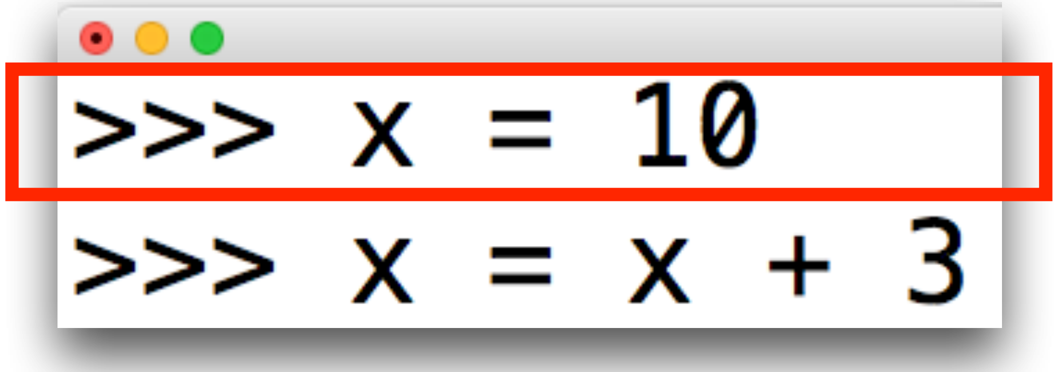
1. Creates object **10** somewhere



```
>>> x = 10
>>> x = x + 3
```

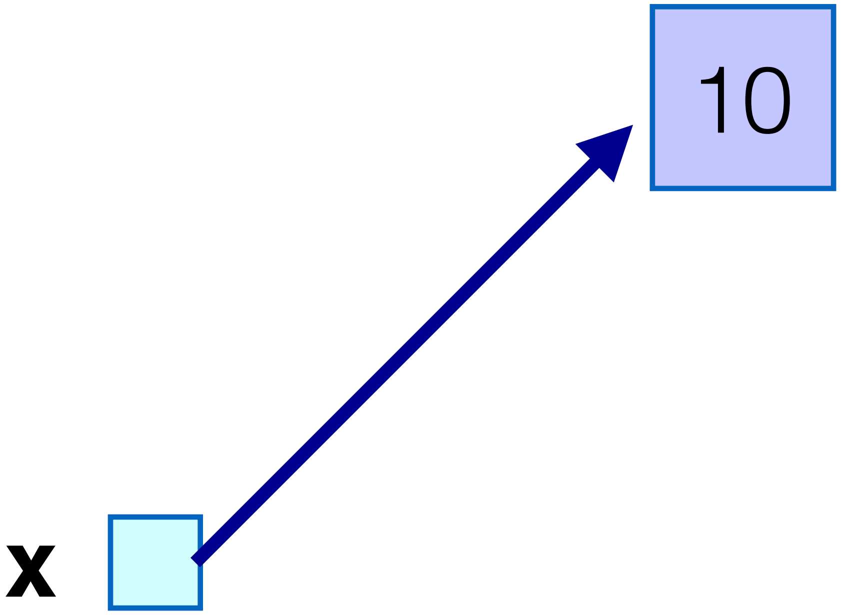
1. Creates object **10** somewhere
2. Creates variable **x**

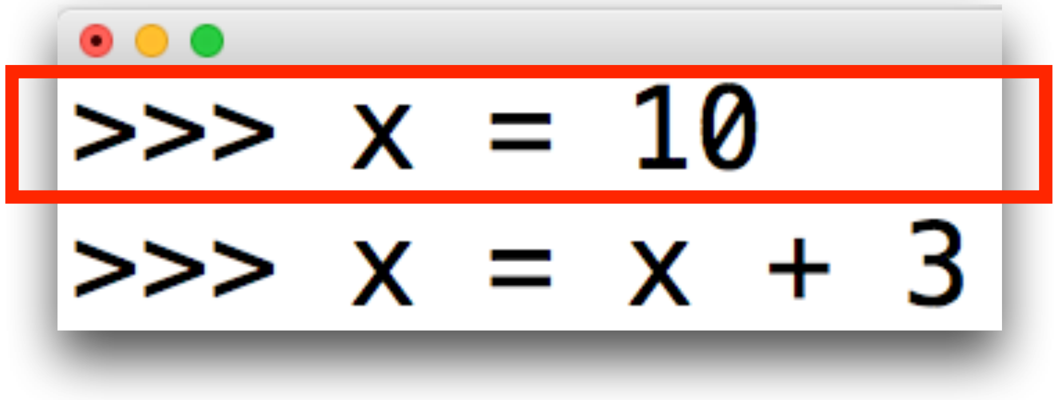




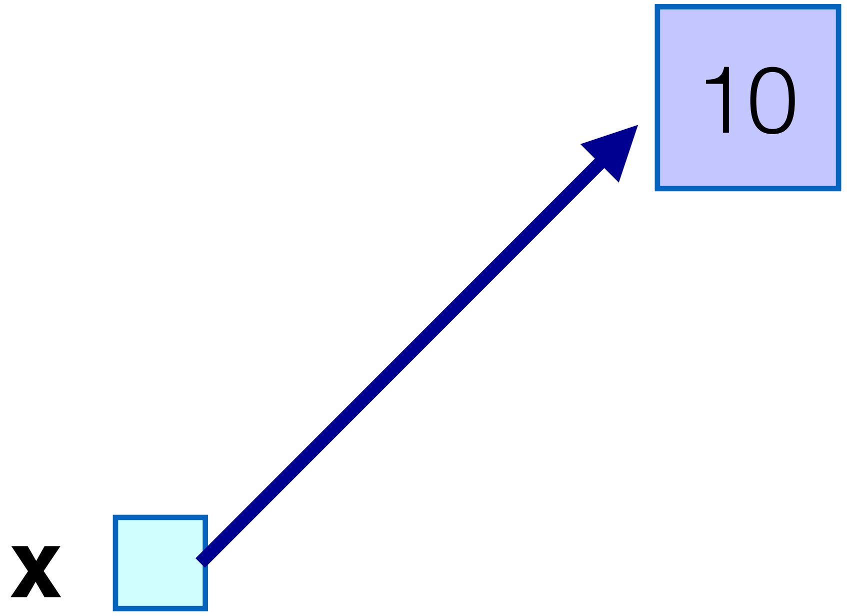
```
>>> x = 10
>>> x = x + 3
```

1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**



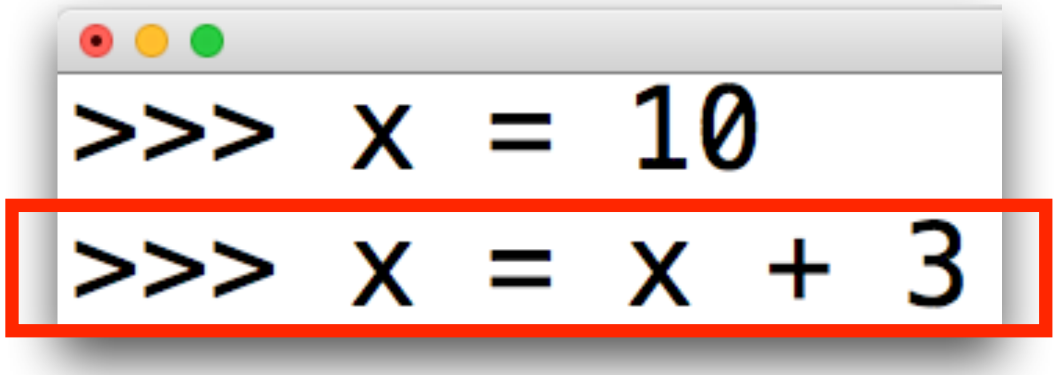


```
>>> x = 10
>>> x = x + 3
```



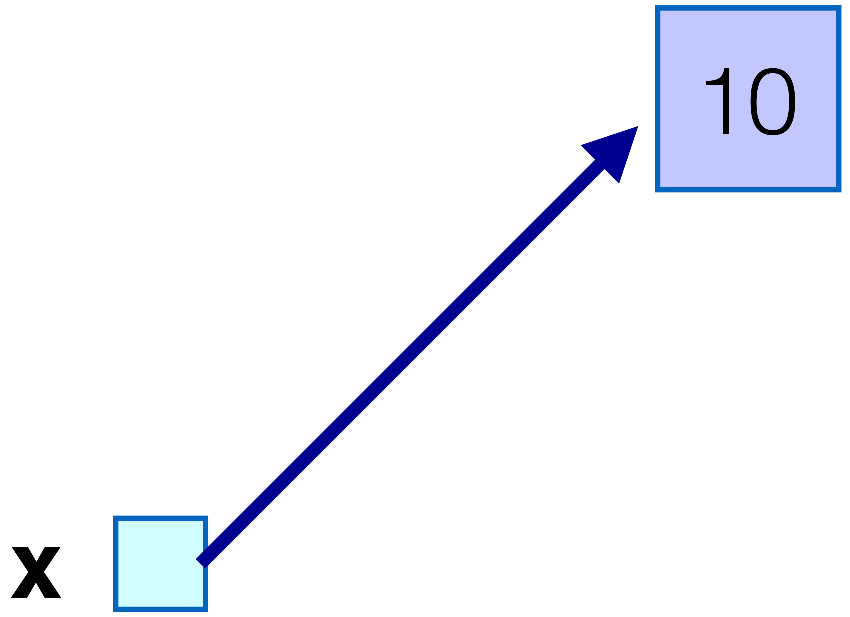
1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**

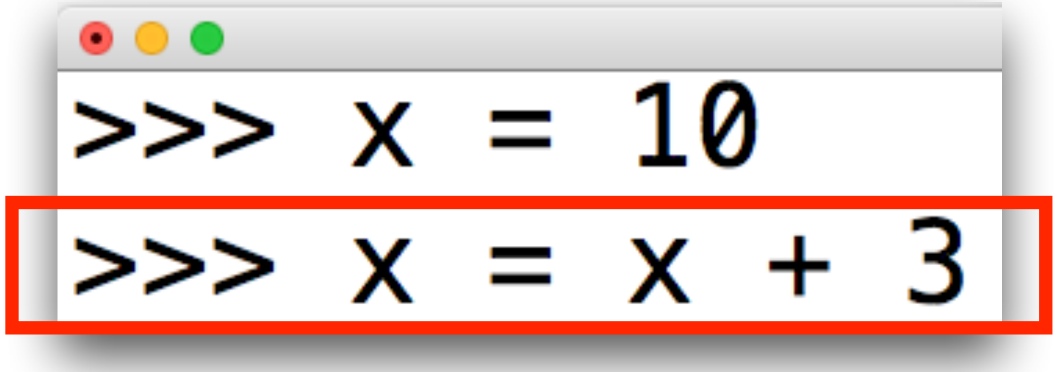
This is why you **must** assign a value to a variable before using it!



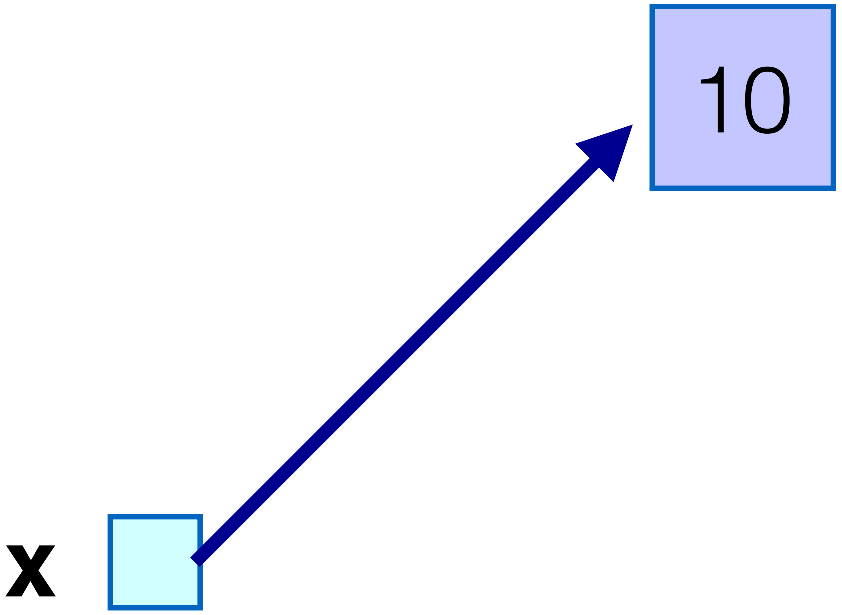
```
>>> x = 10
>>> x = x + 3
```

1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Evaluates **x + 3**



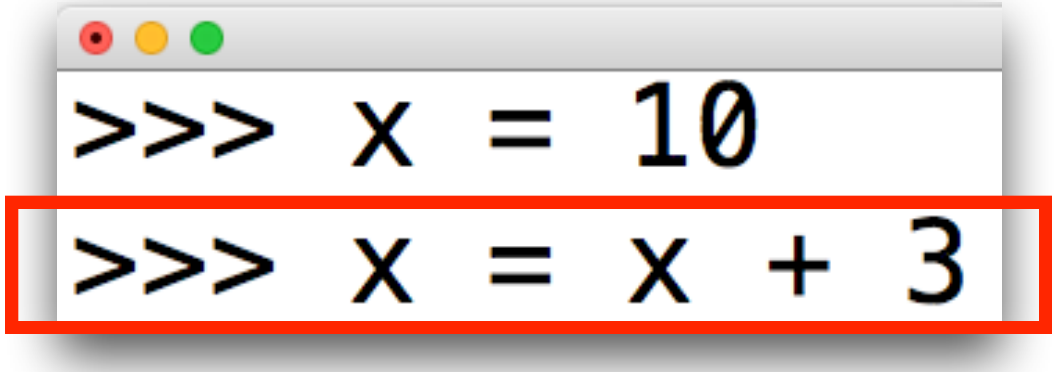


```
>>> x = 10
>>> x = x + 3
```

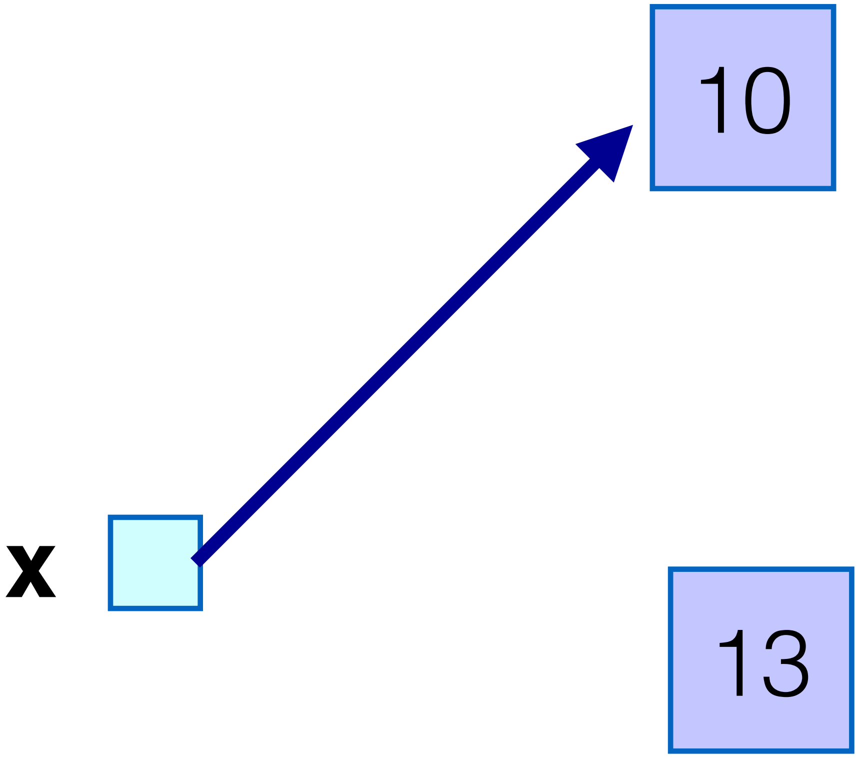


1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Evaluates **x + 3**

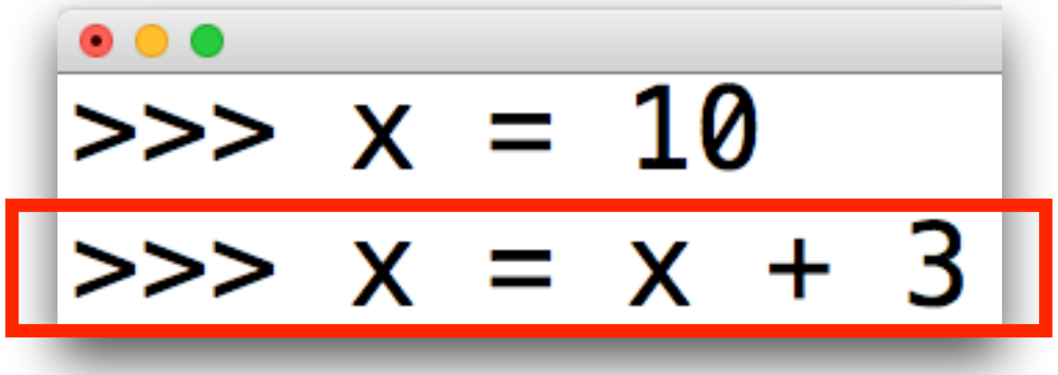
A variable in an expression is immediately **replaced** with the object it currently refers to. Then the expression is evaluated.



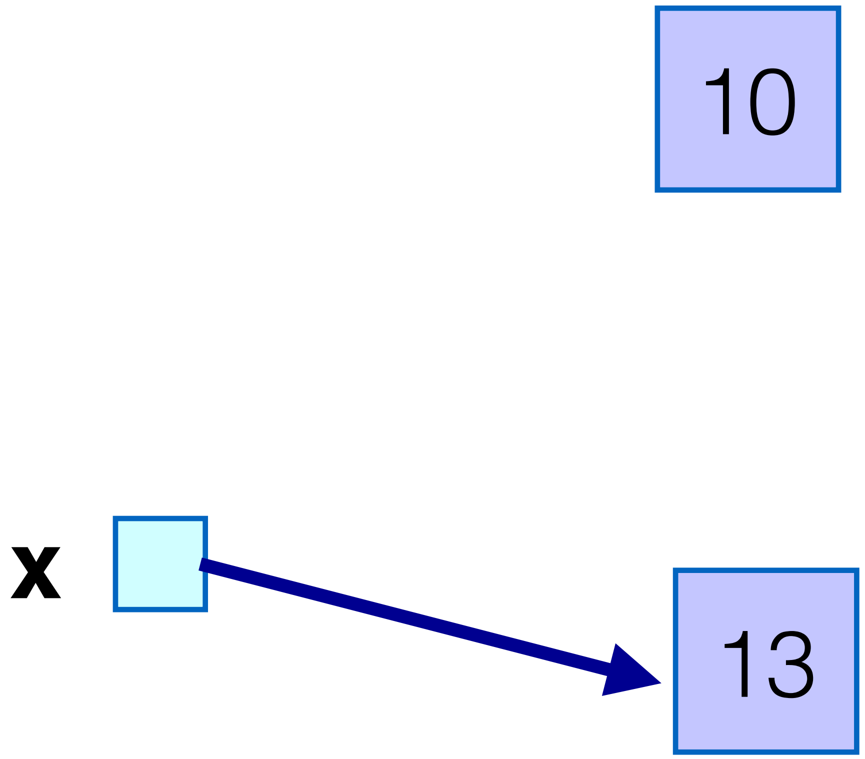
```
>>> x = 10
>>> x = x + 3
```



1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Evaluates **x + 3**
5. Creates object **13**

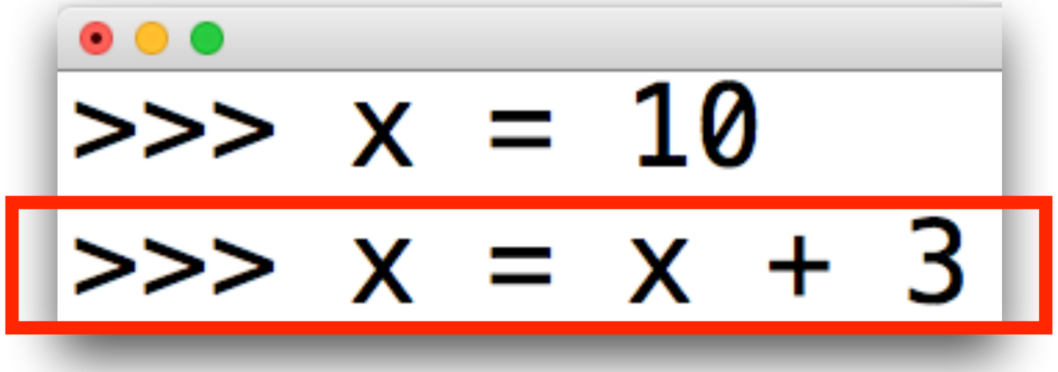


```
>>> x = 10
>>> x = x + 3
```

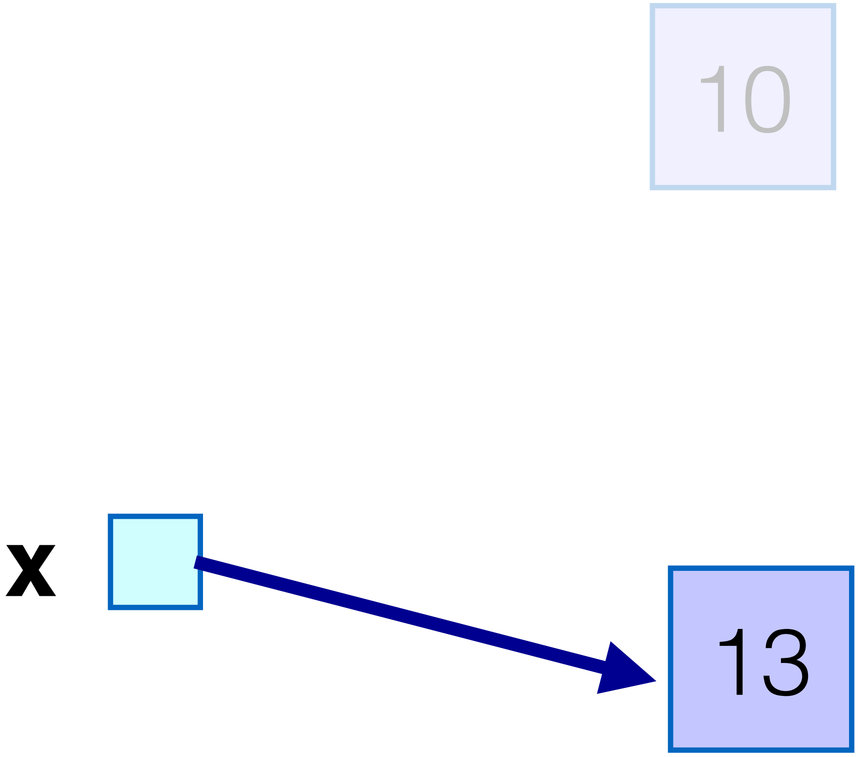


1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Evaluates **x + 3**
5. Creates object **13**
6. Links **x** to **13**

Garbage collection:
Automatically removes objects
that are not referenced.

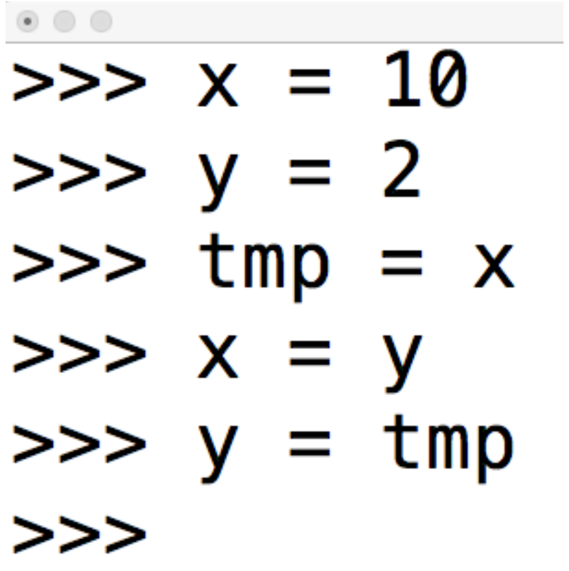


```
>>> x = 10
>>> x = x + 3
```



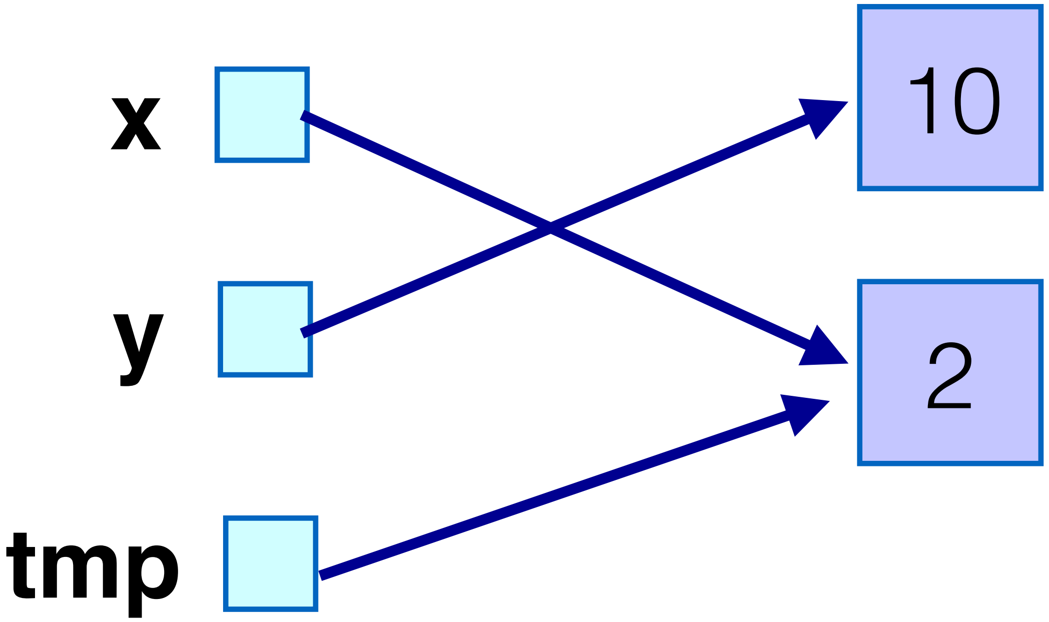
1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Evaluates **x + 3**
5. Creates object **13**
6. Links **x** to **13**

After the following:



```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

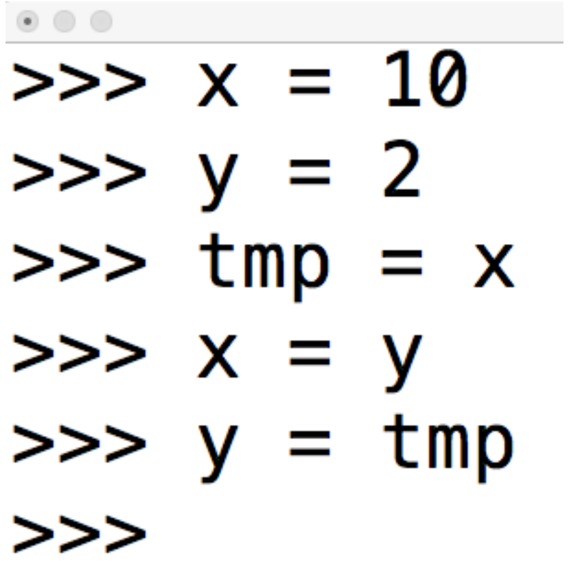
We have:



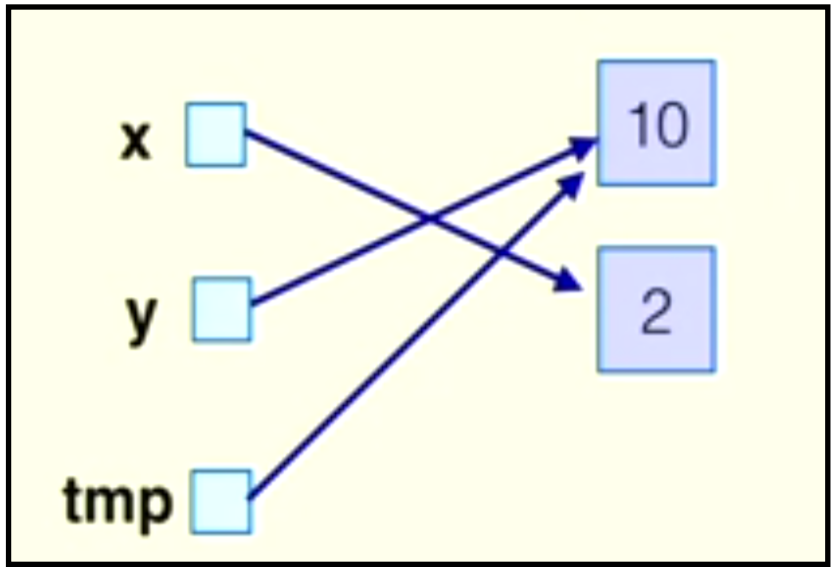
A) True.

B) False.

After the following:

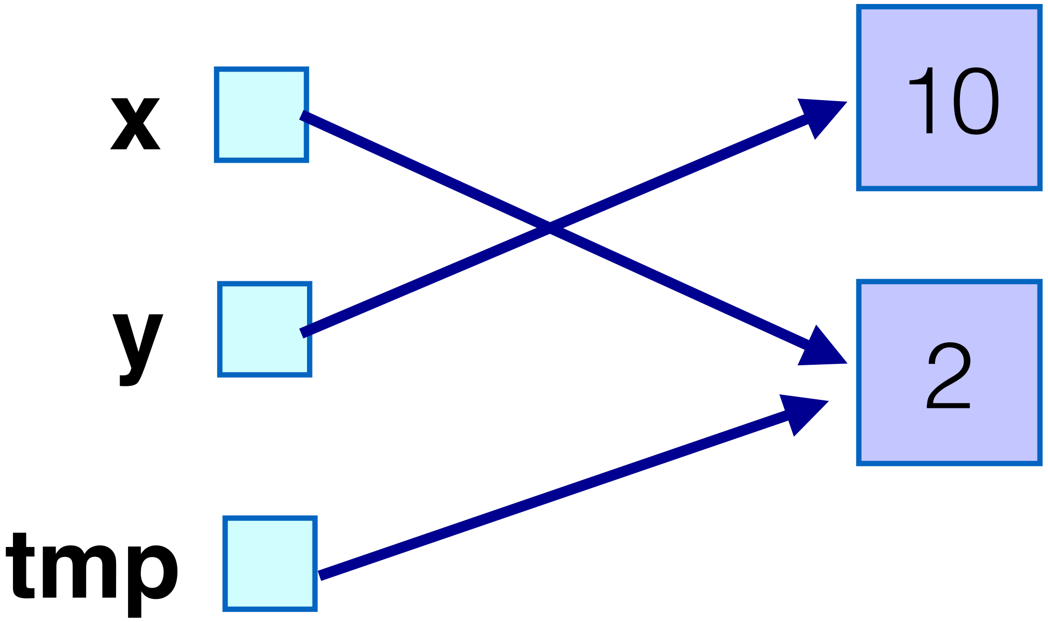


```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```



We will see why in a minute…

We have:

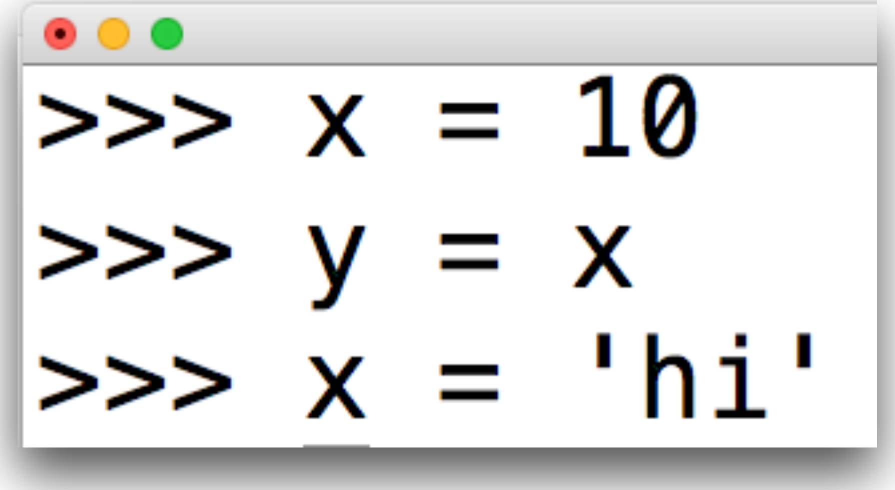


A) True.

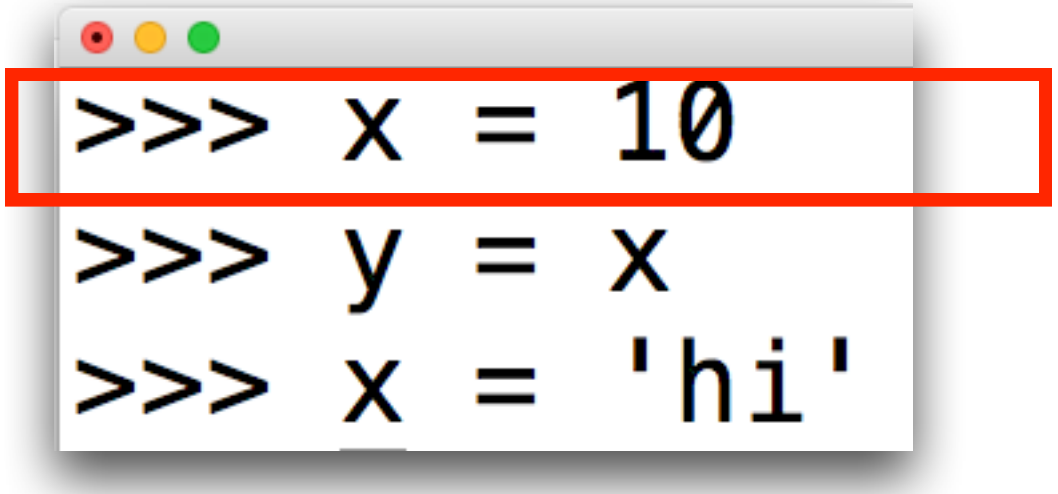
B) **False**.

- Every time a new value is created, Python creates a new object (a chunk of memory) to represent it.

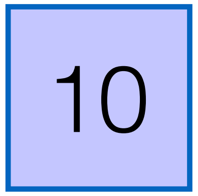
- **What about assigning a variable to another variable?**



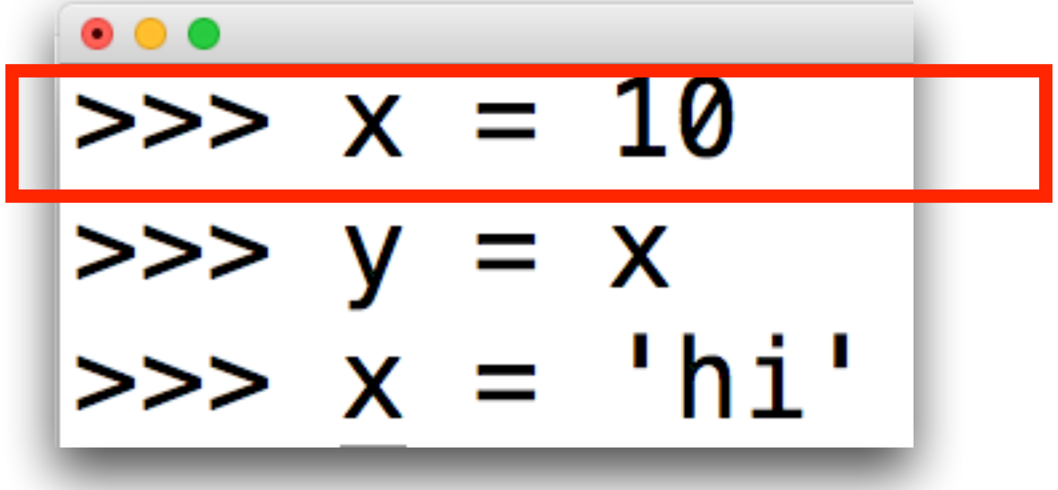
```
>>> x = 10
>>> y = x
>>> x = 'hi'
```



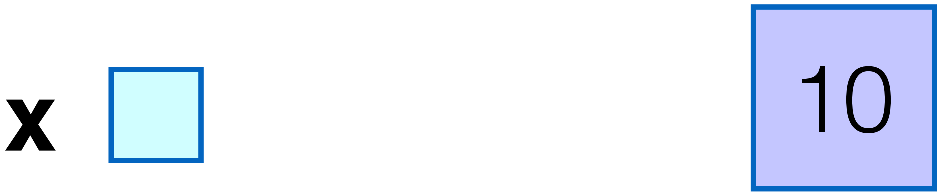
```
>>> x = 10
>>> y = x
>>> x = 'hi'
```



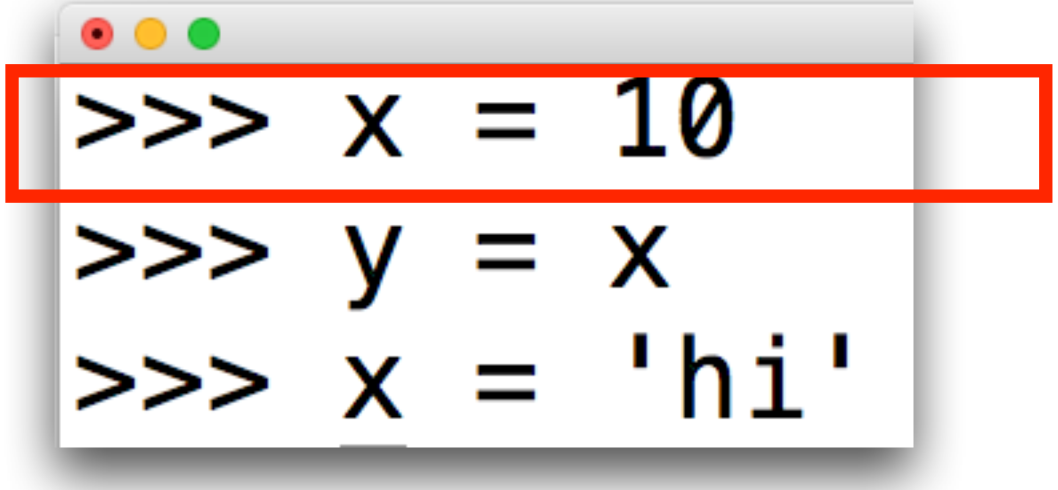
1. Creates object **10** somewhere



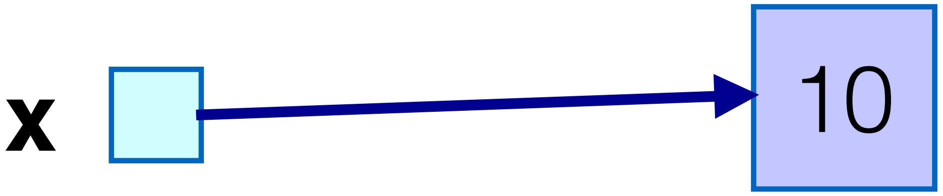
```
>>> x = 10
>>> y = x
>>> x = 'hi'
```



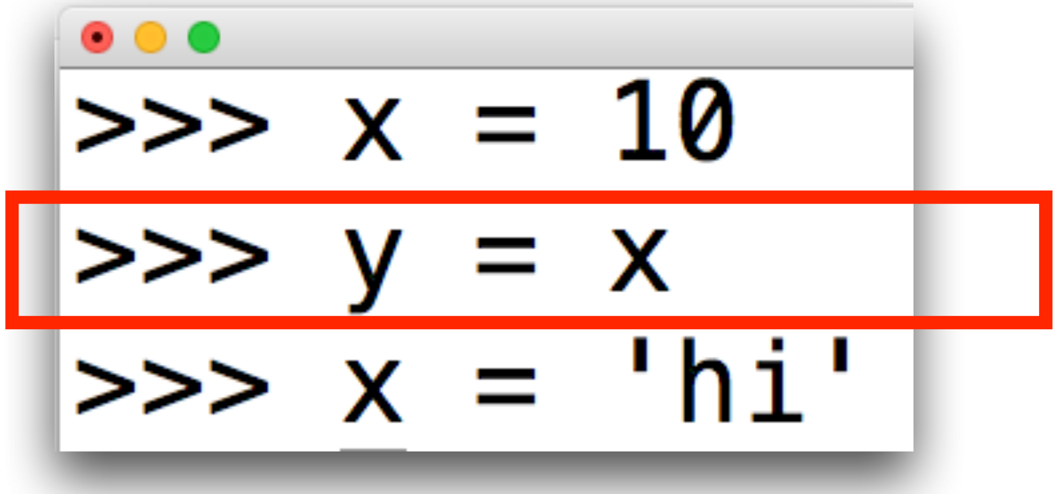
1. Creates object **10** somewhere
2. Creates variable **x**



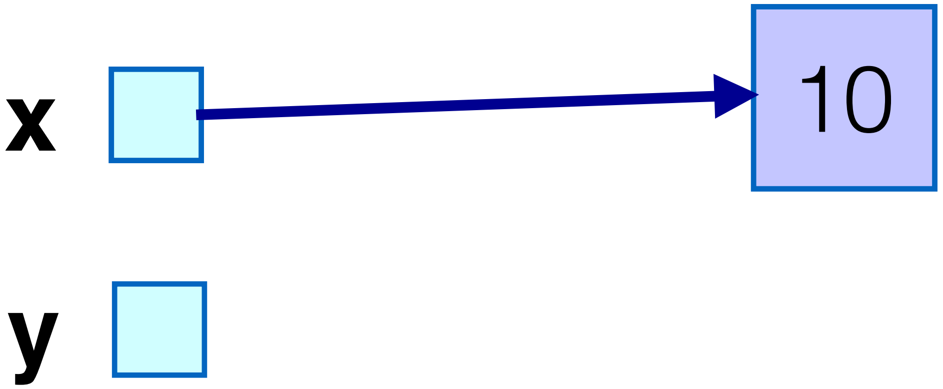
```
>>> x = 10
>>> y = x
>>> x = 'hi'
```



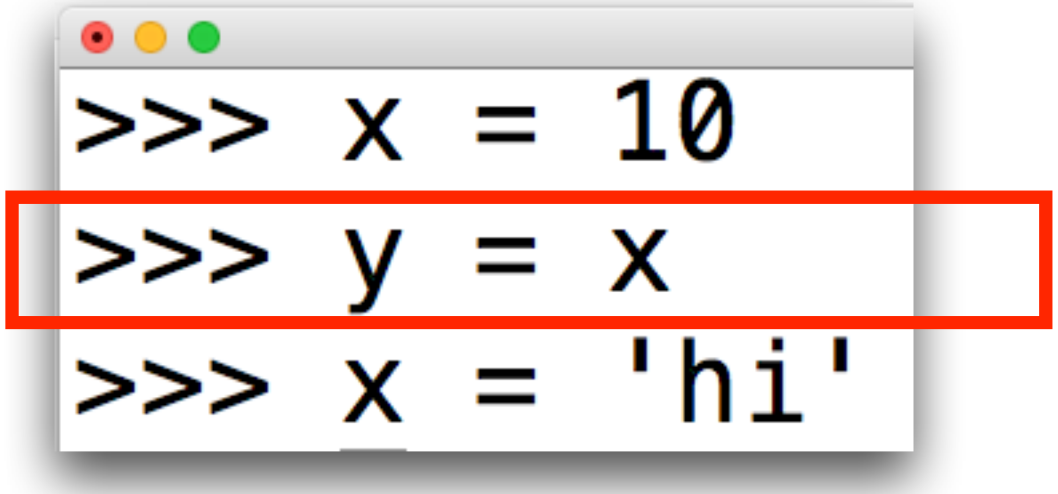
1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**



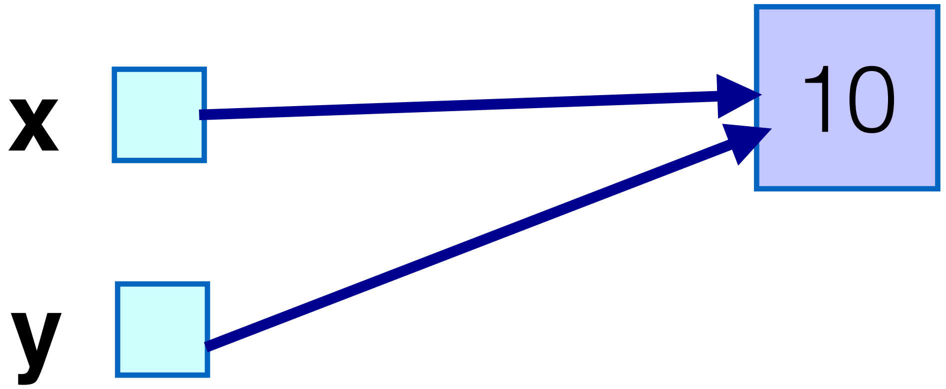
```
>>> x = 10
>>> y = x
>>> x = 'hi'
```



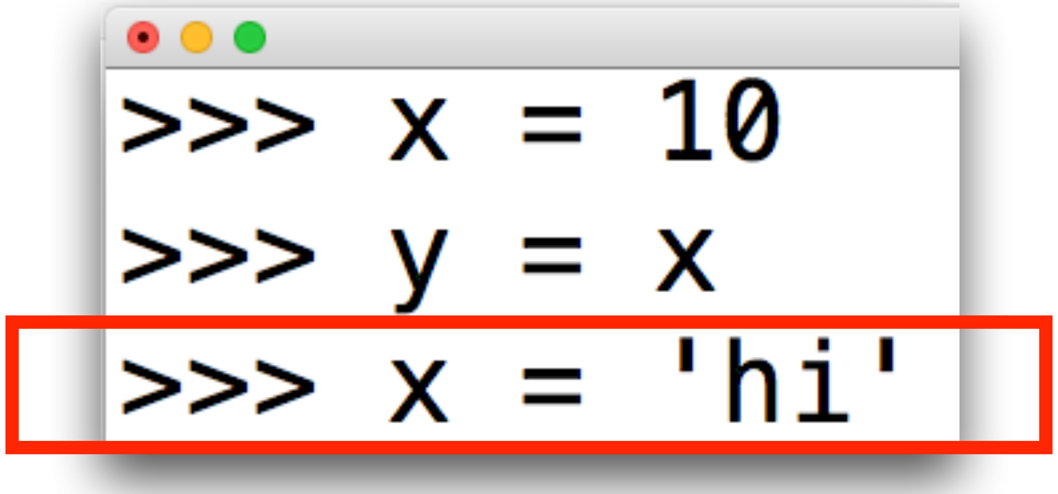
1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Creates variable **y**



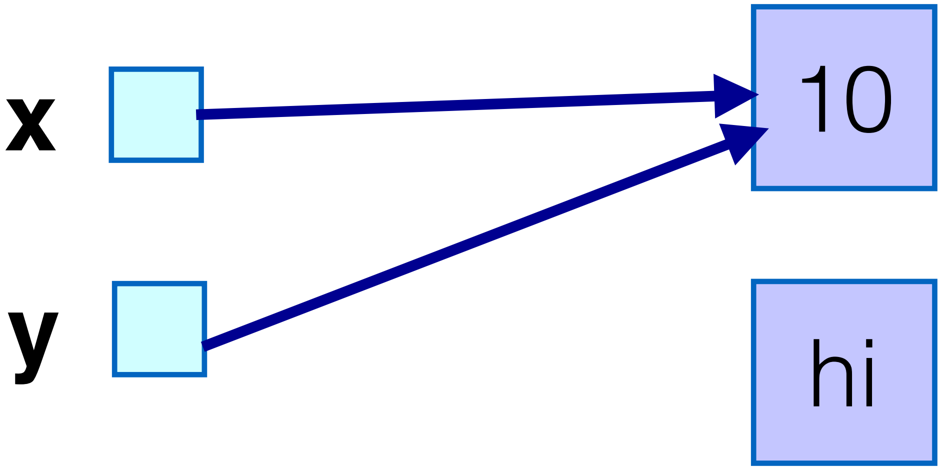
```
>>> x = 10
>>> y = x
>>> x = 'hi'
```



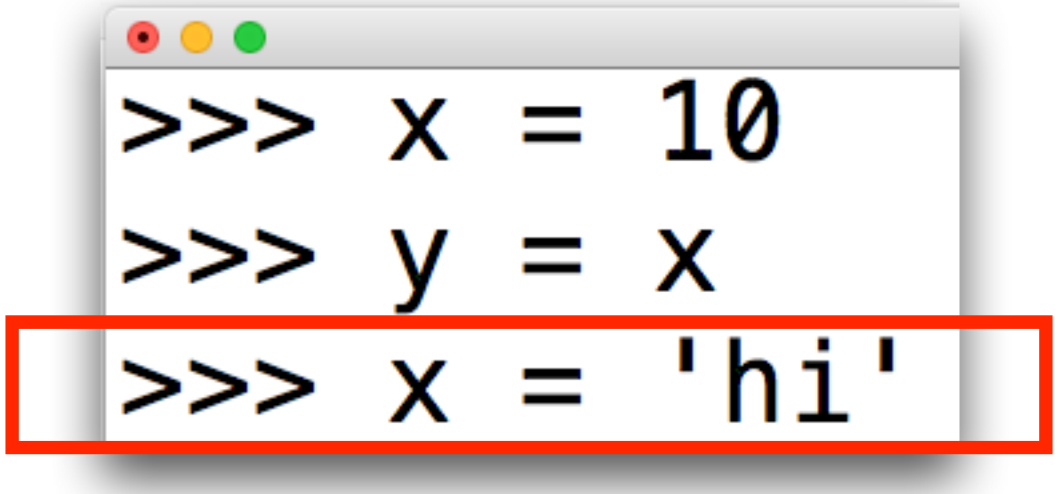
1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Creates variable **y**
5. Links it to the object pointed to by **x**



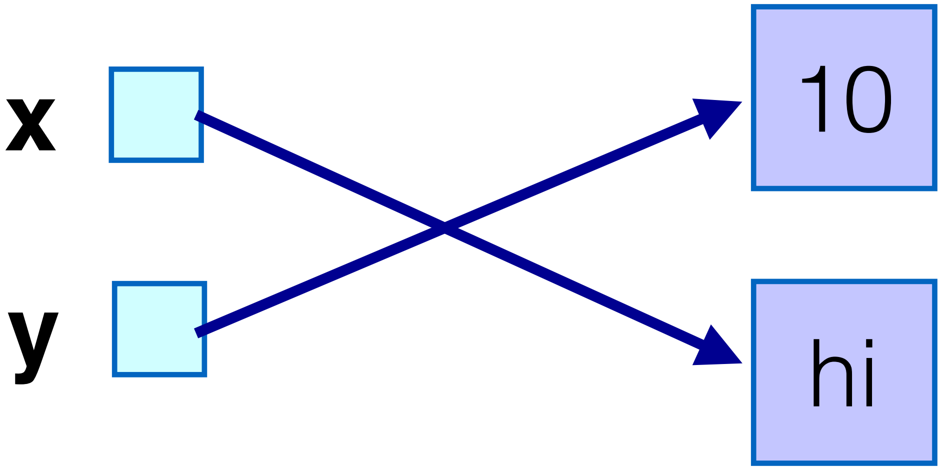
```
>>> x = 10
>>> y = x
>>> x = 'hi'
```



1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Creates variable **y**
5. Links it to the object pointed to by **x**
6. Creates the object **‘hi’**

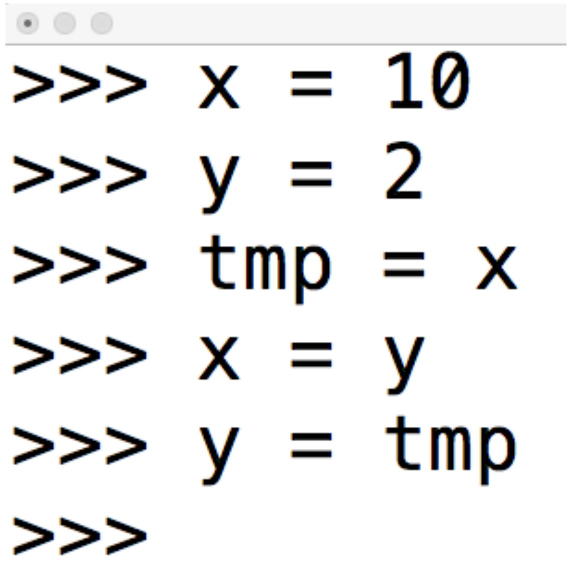


```
>>> x = 10
>>> y = x
>>> x = 'hi'
```



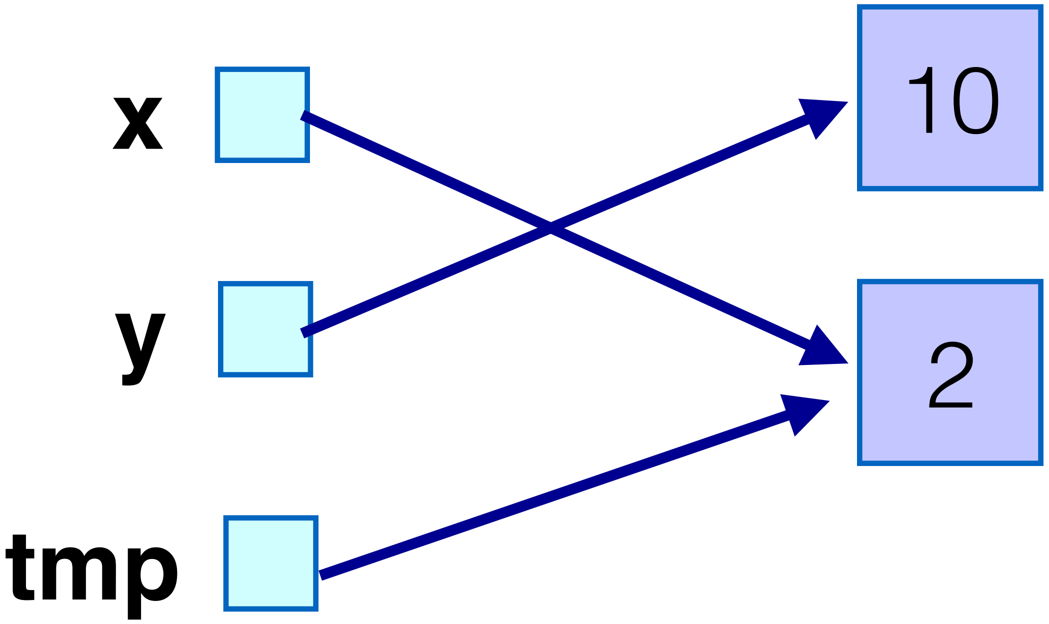
1. Creates object **10** somewhere
2. Creates variable **x**
3. Links **x** to **10**
4. Creates variable **y**
5. Links it to the object pointed to by **x**
6. Creates the object **‘hi’**
7. Links **x** to this object.

After the following:



```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

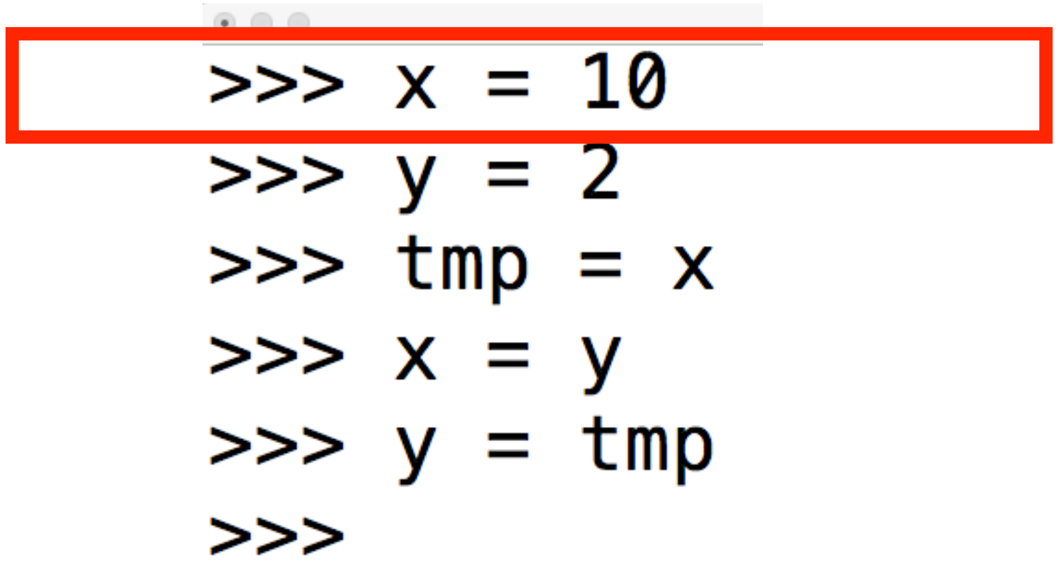
We have:



A) True.

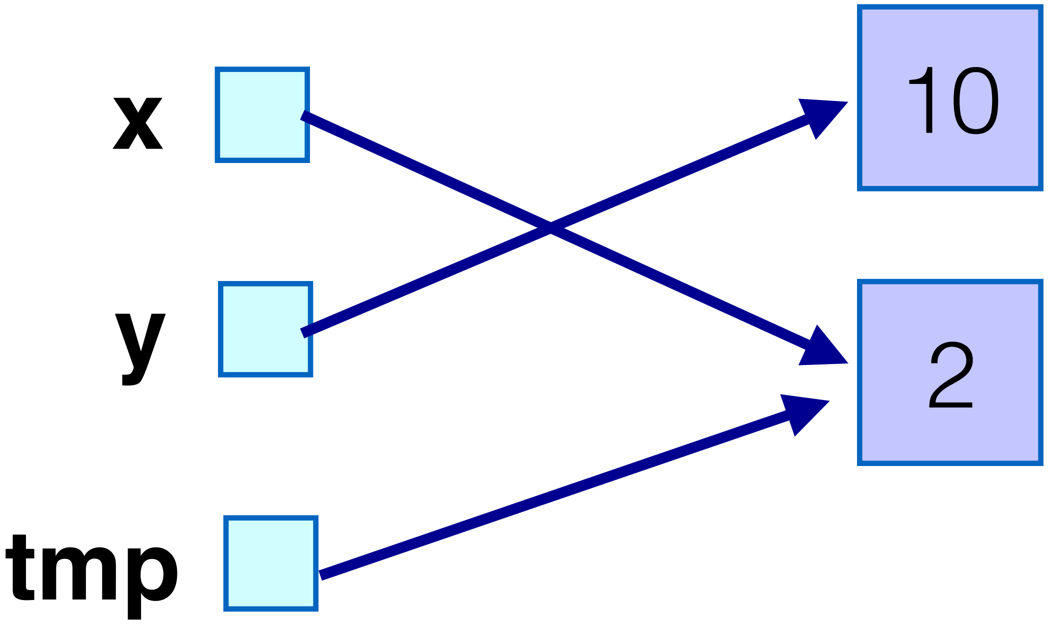
B) False.

After the following:



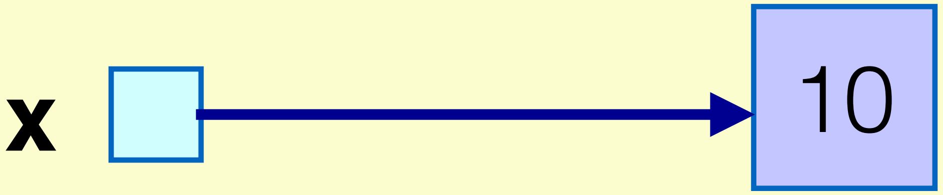
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

We have:

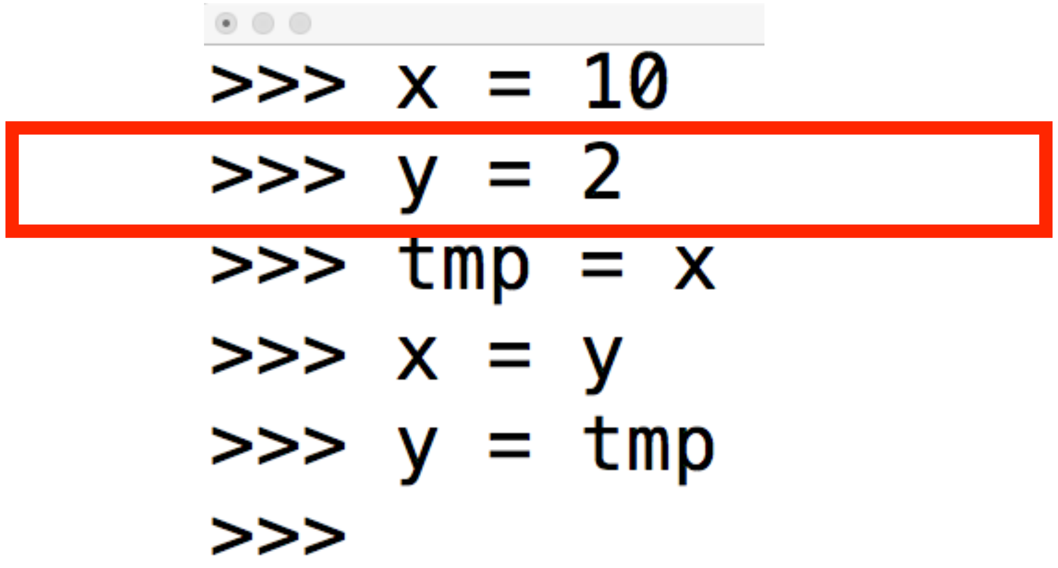


A) True.

B) False.

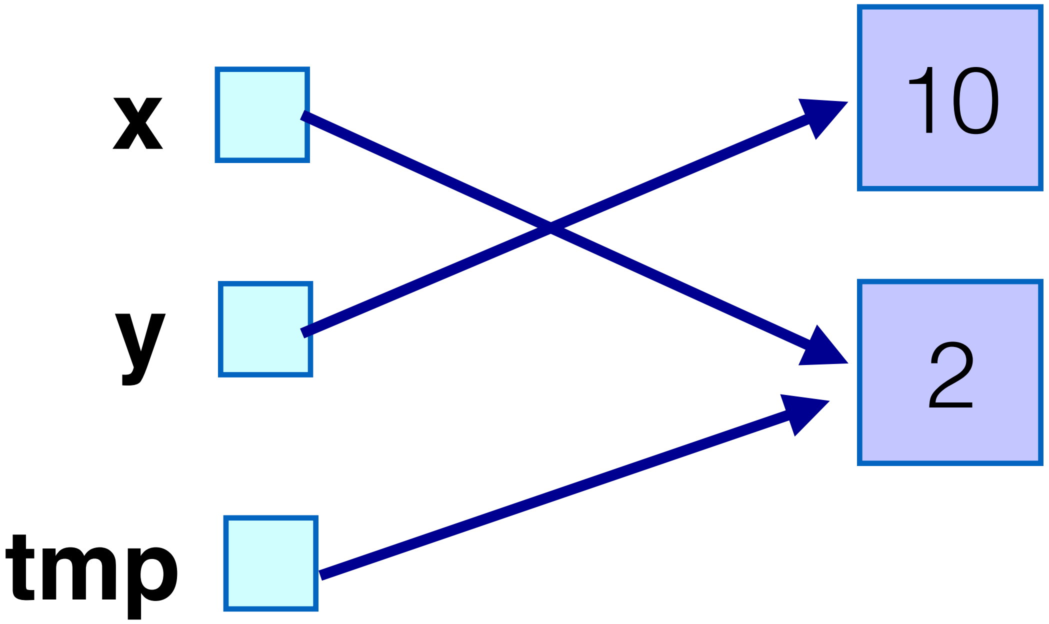


After the following:



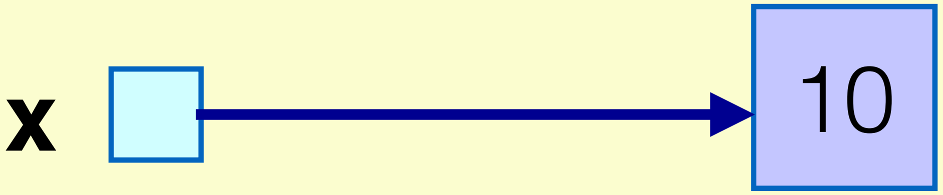
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

We have:

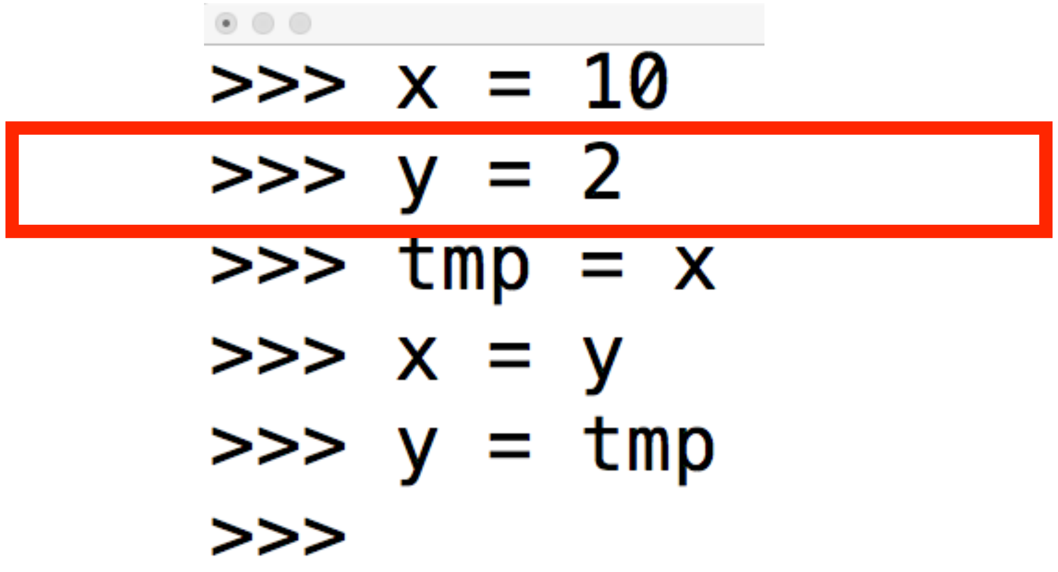


A) True.

B) False.

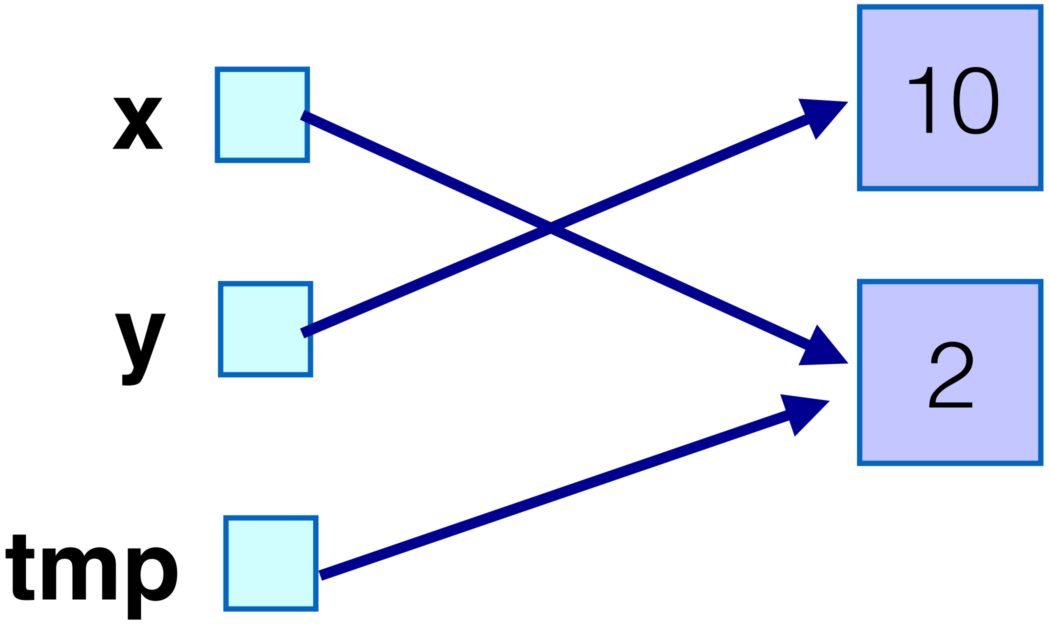


After the following:



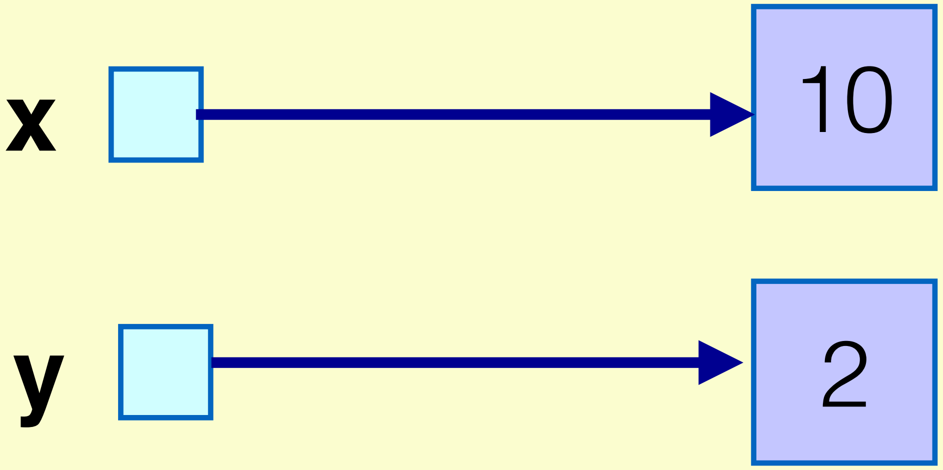
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

We have:

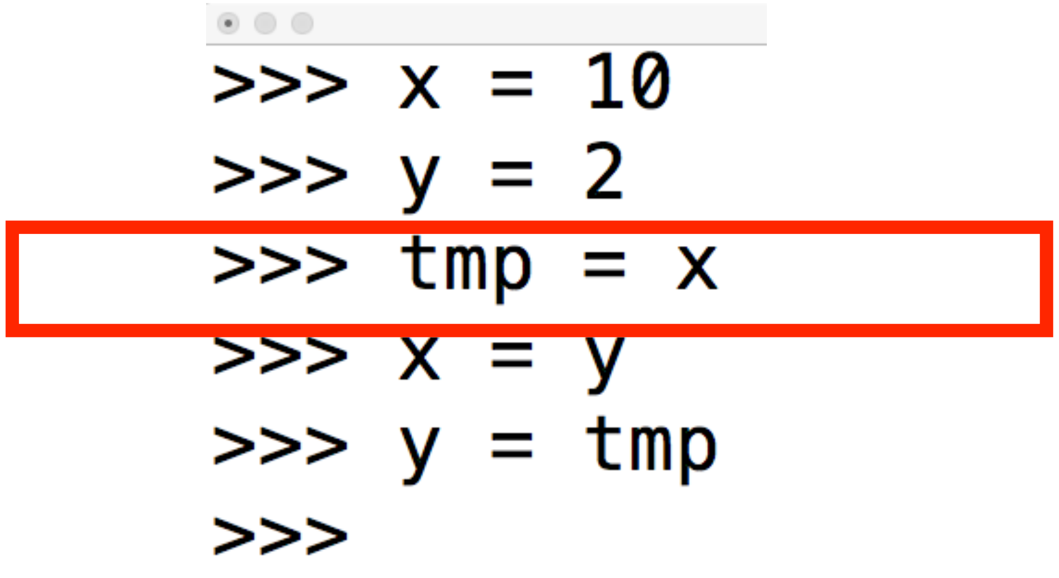


A) True.

B) False.

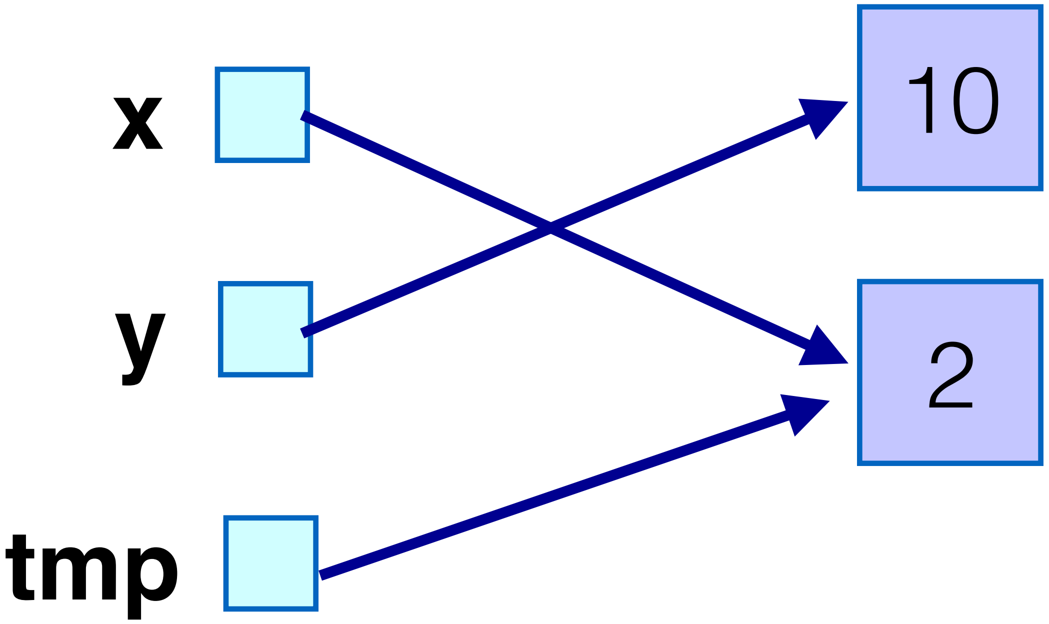


After the following:



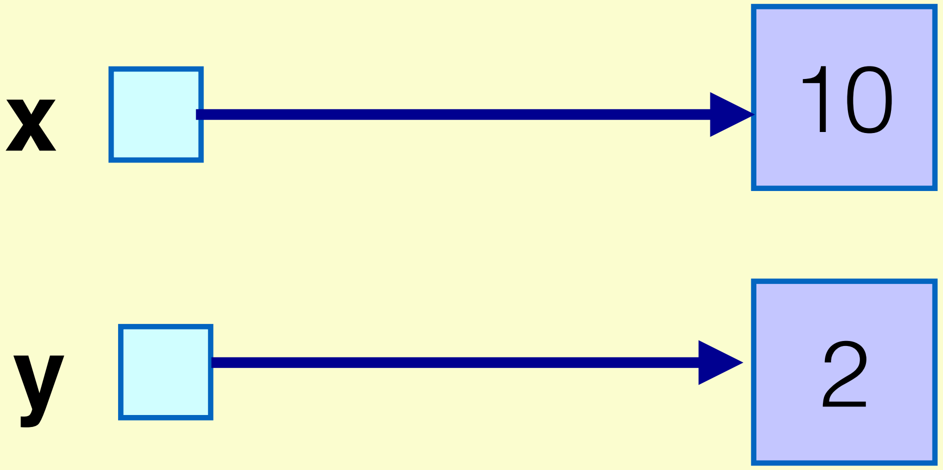
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

We have:



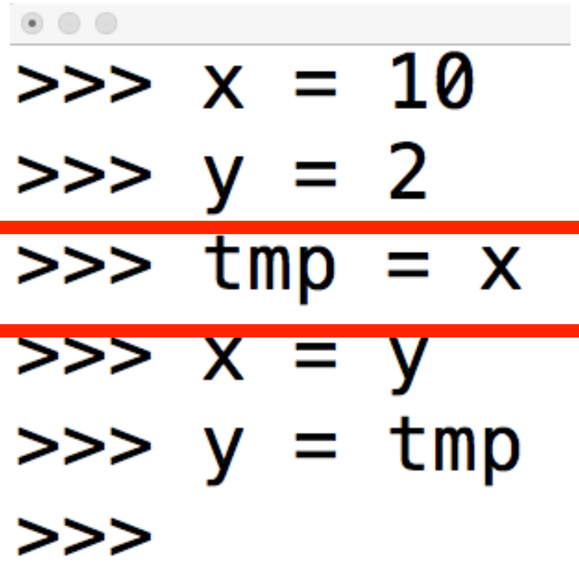
A) True.

B) False.

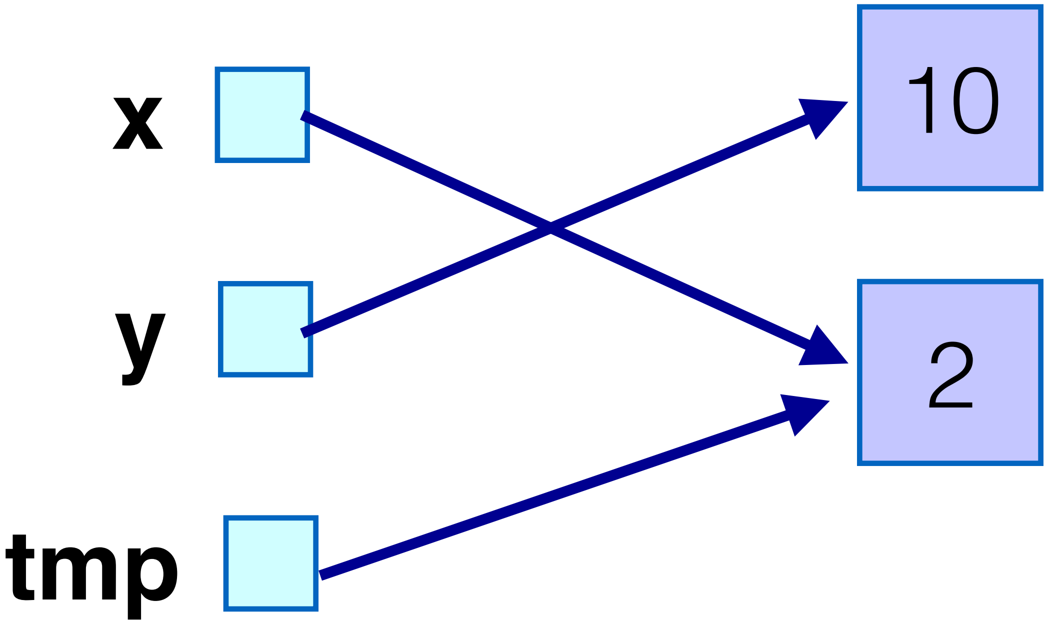


After the following:

We have:

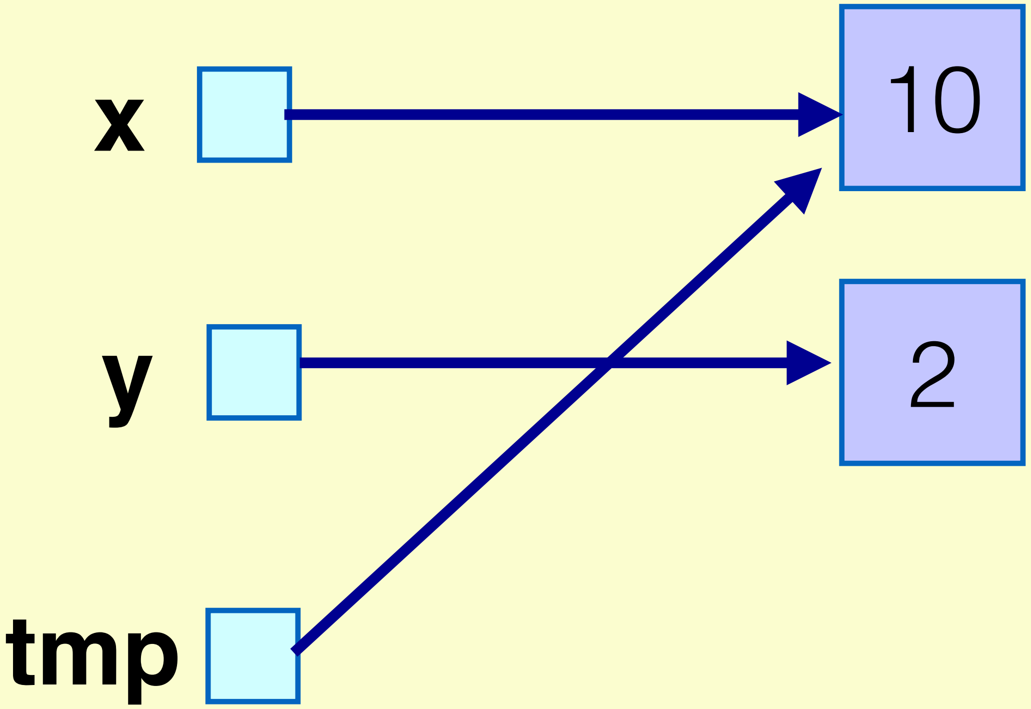


```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

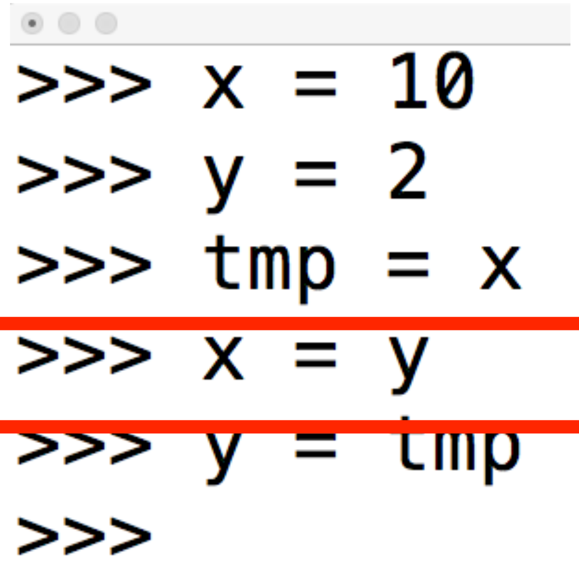


A) True.

B) False.

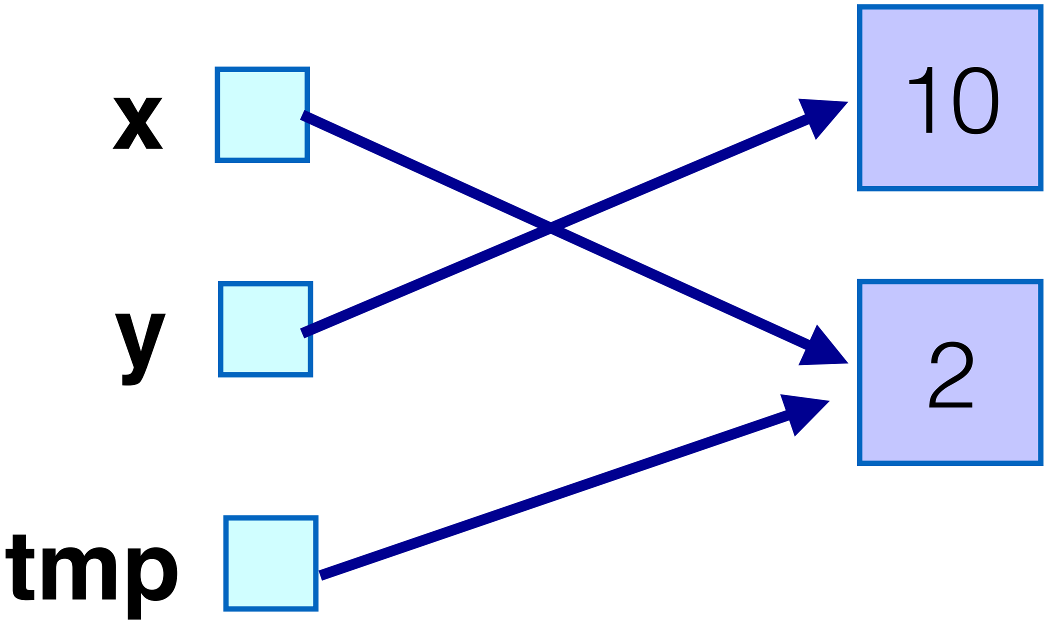


After the following:



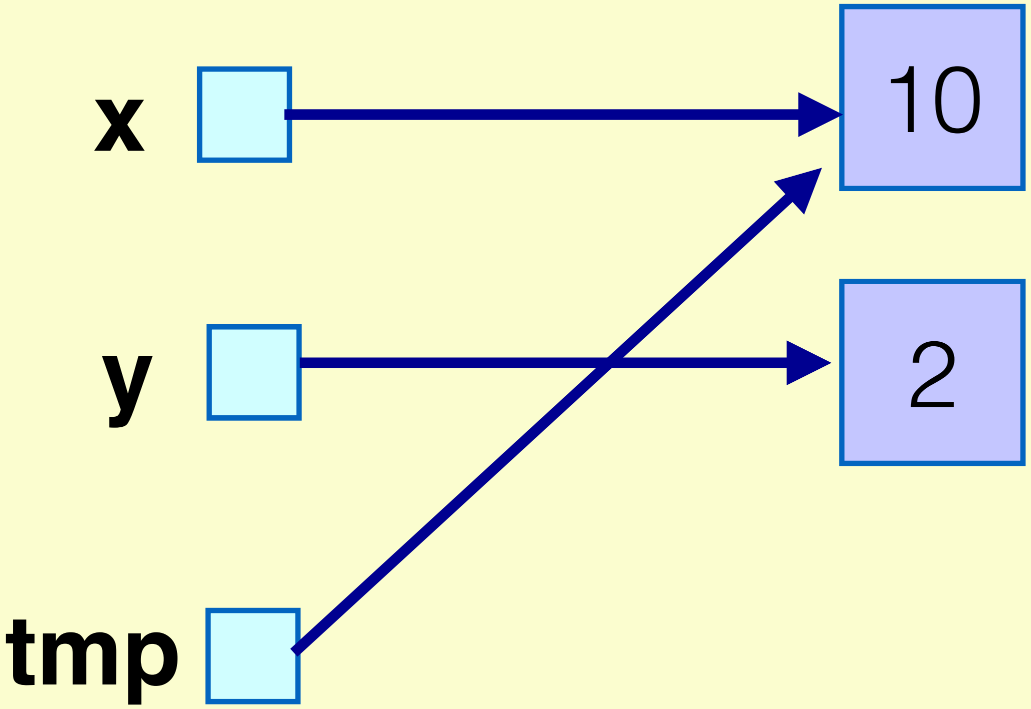
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

We have:

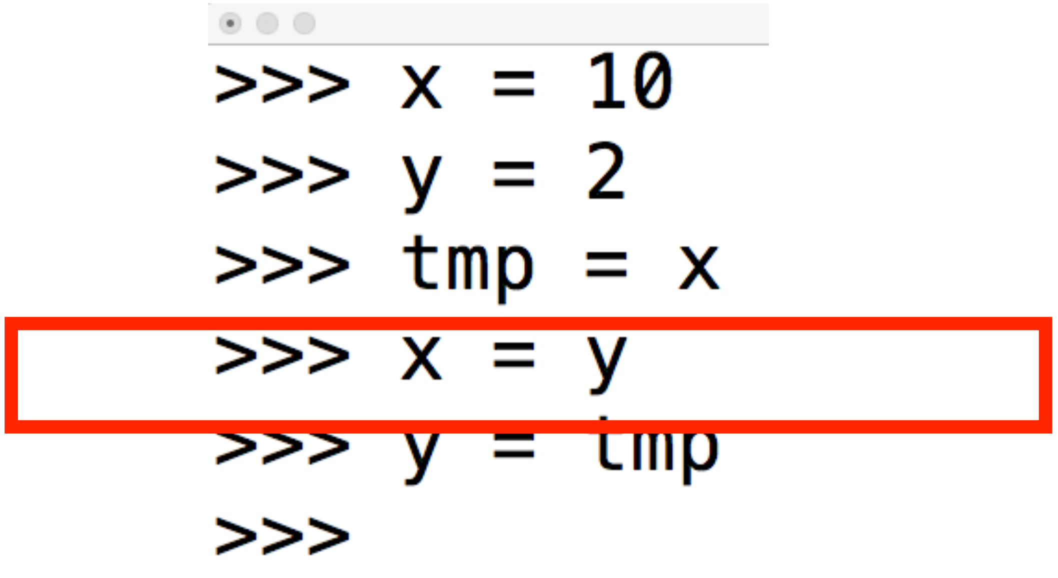


A) True.

B) False.

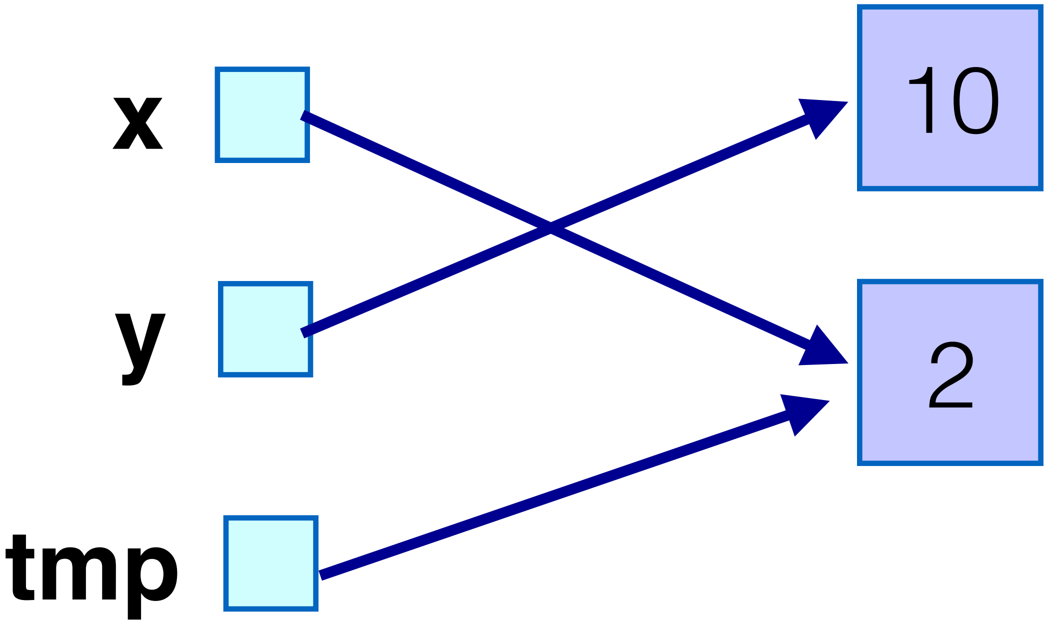


After the following:



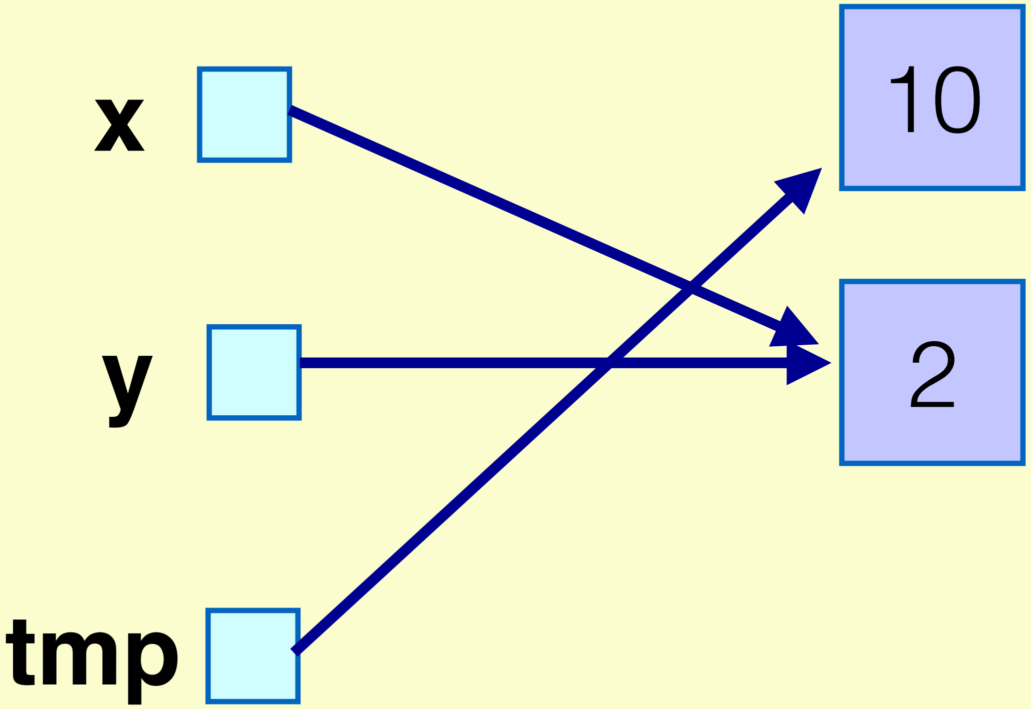
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

We have:

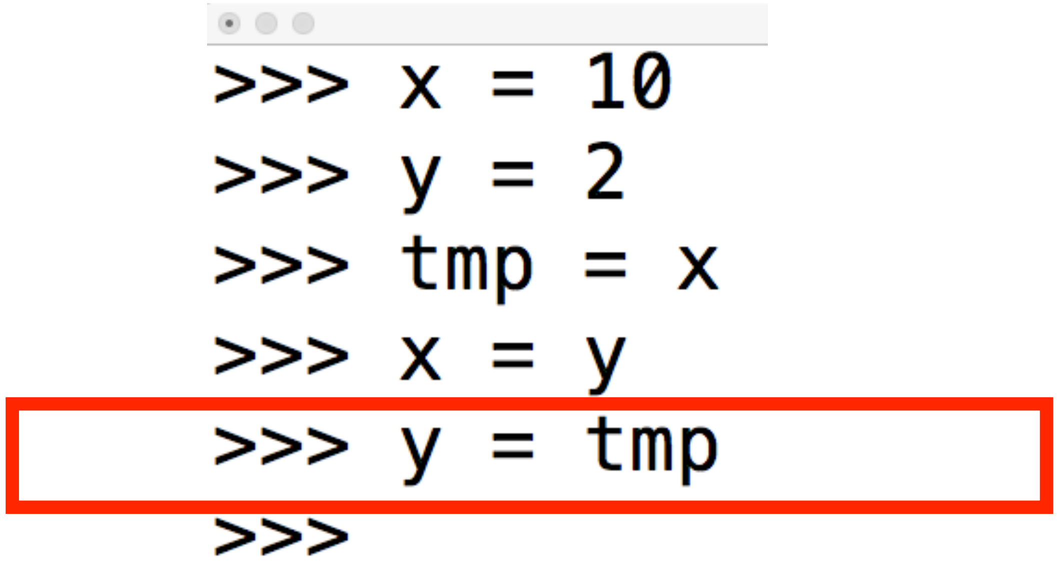


A) True.

B) False.

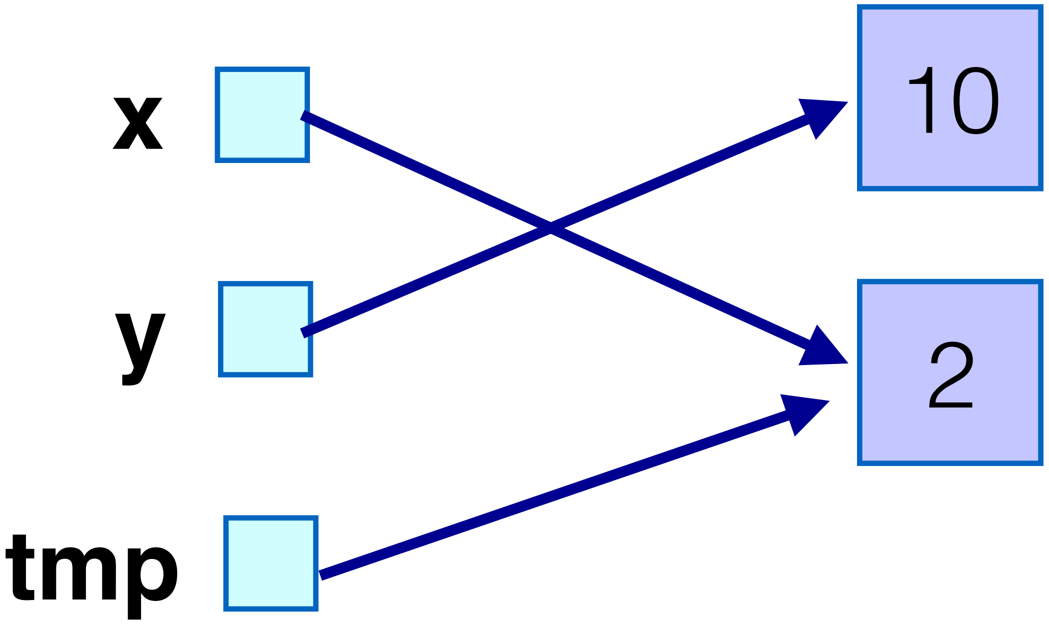


After the following:



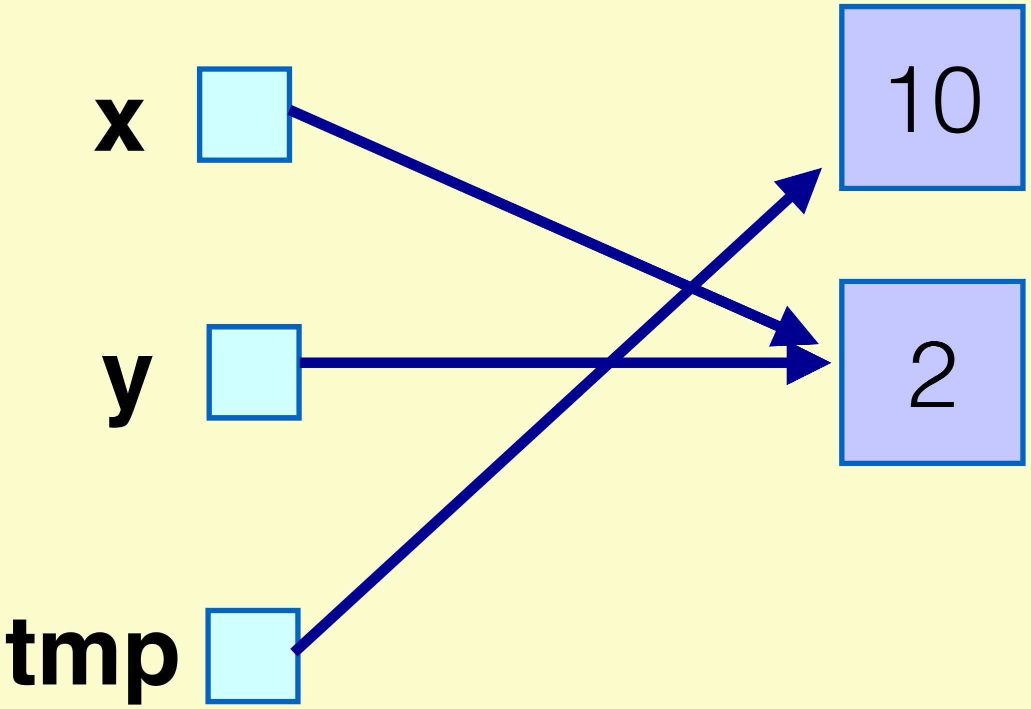
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

We have:

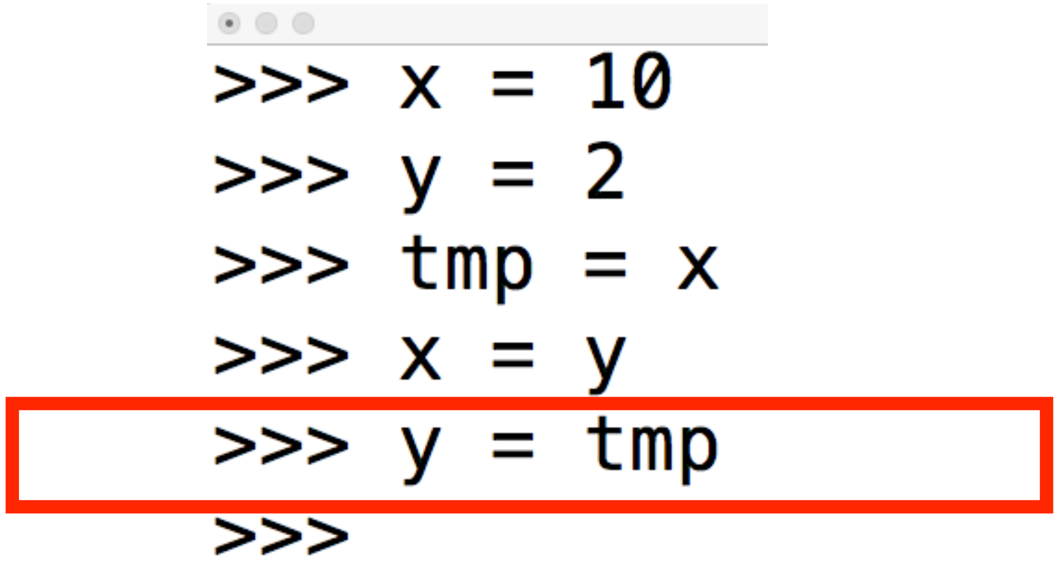


A) True.

B) False.

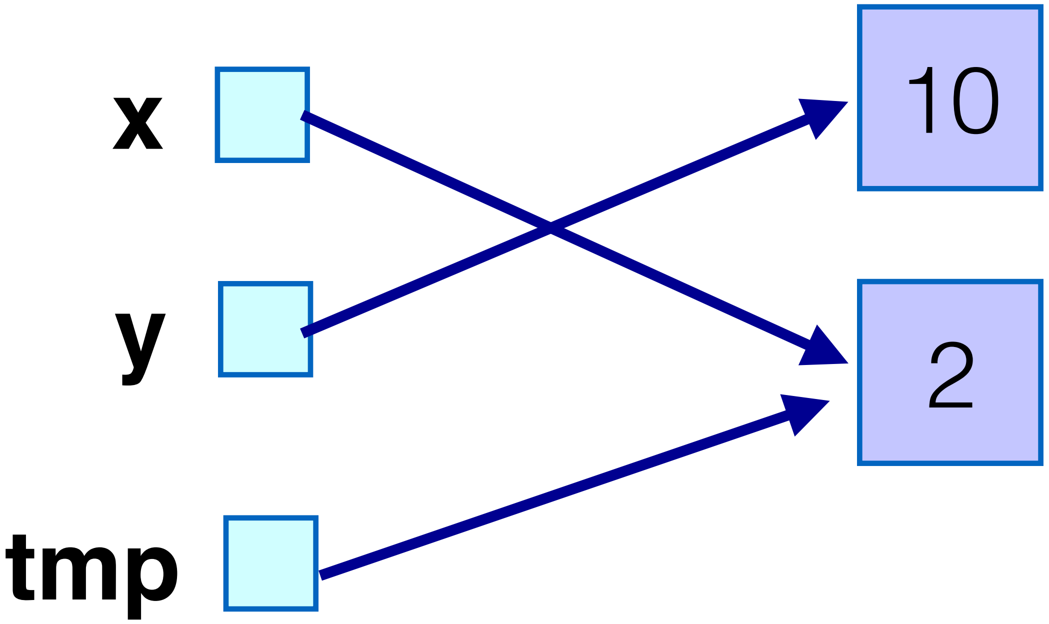


After the following:



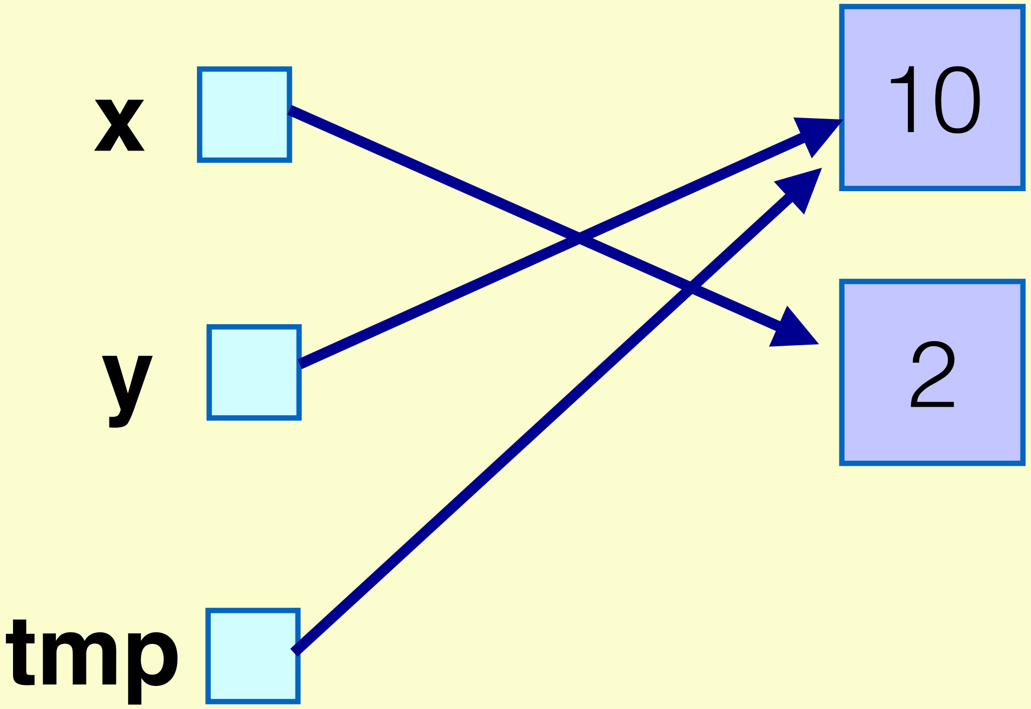
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

We have:

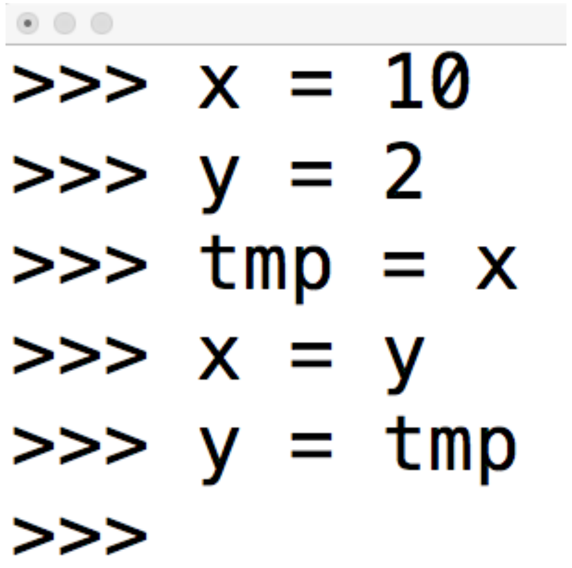


A) True.

B) False.

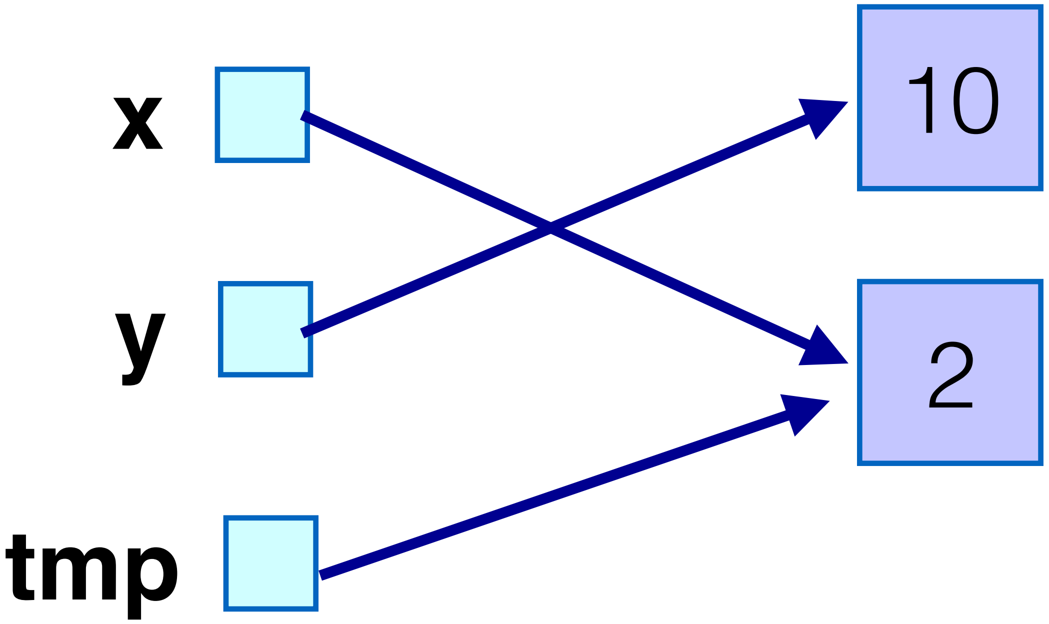


After the following:



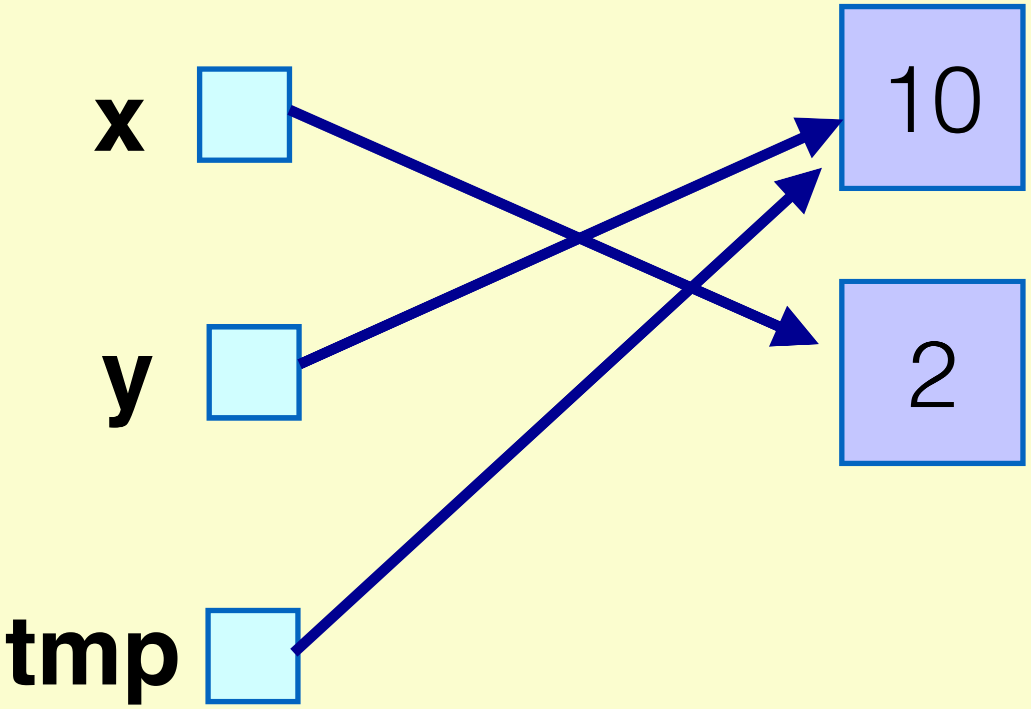
```
>>> x = 10
>>> y = 2
>>> tmp = x
>>> x = y
>>> y = tmp
>>>
```

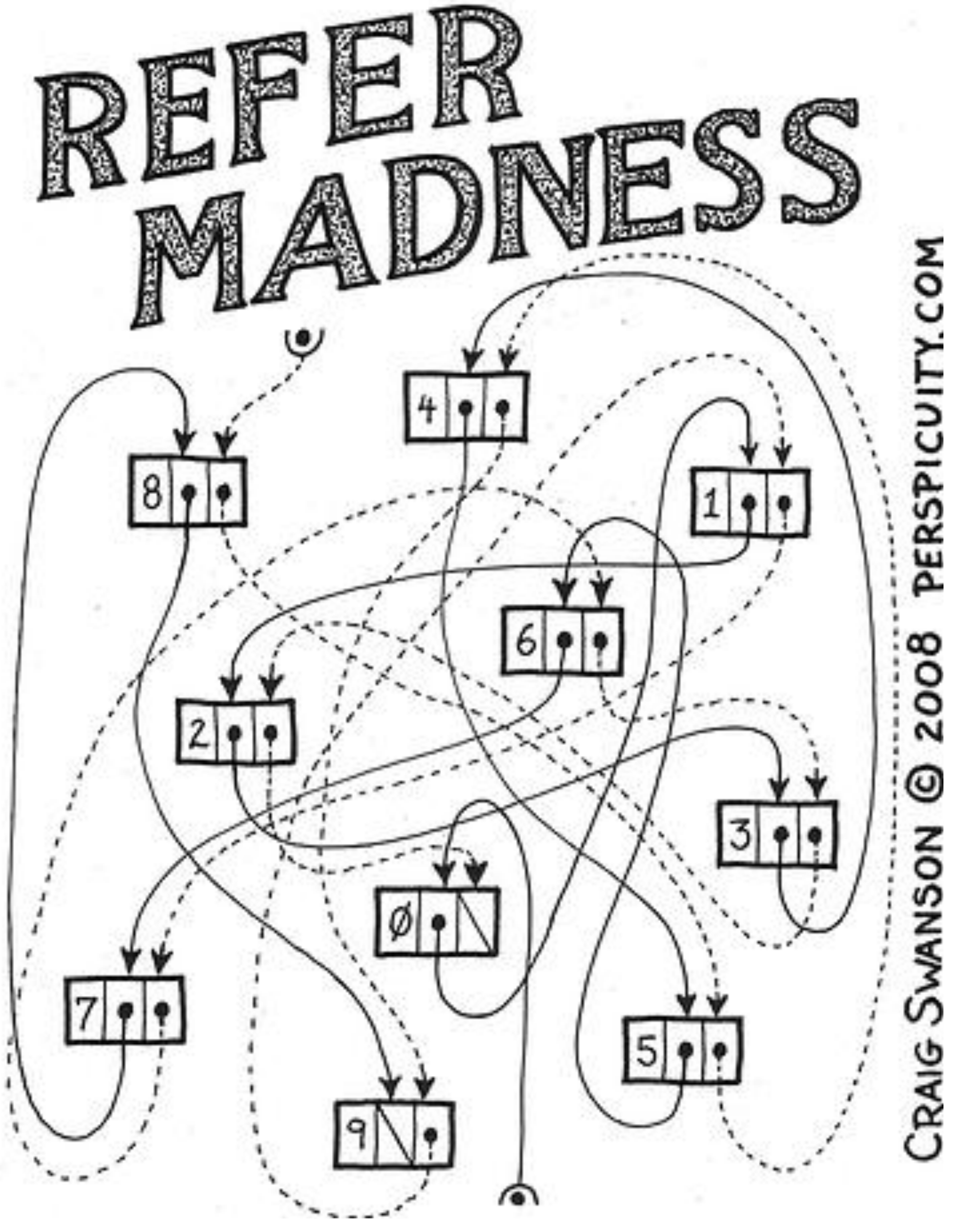
We have:



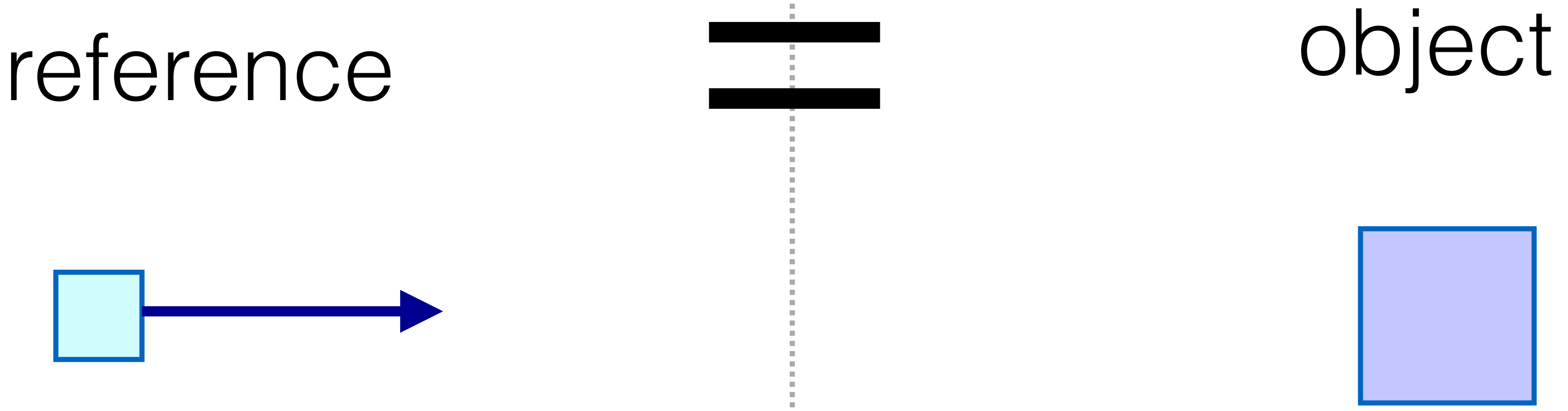
A) True.

B) **False**.





Source: perspicuity.com



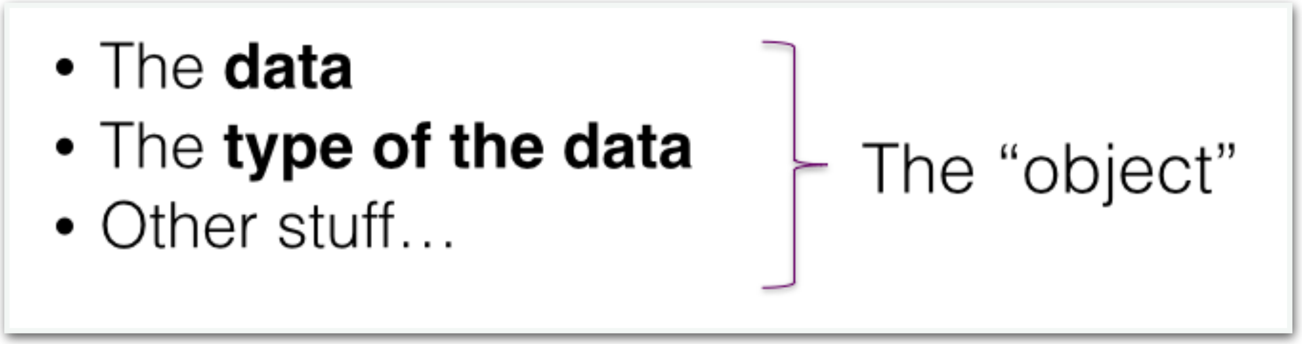
reference = object

What about Python lists?

- How are Python lists represented internally?
    - Remember: they are arrays.
    - But they are also **objects**.

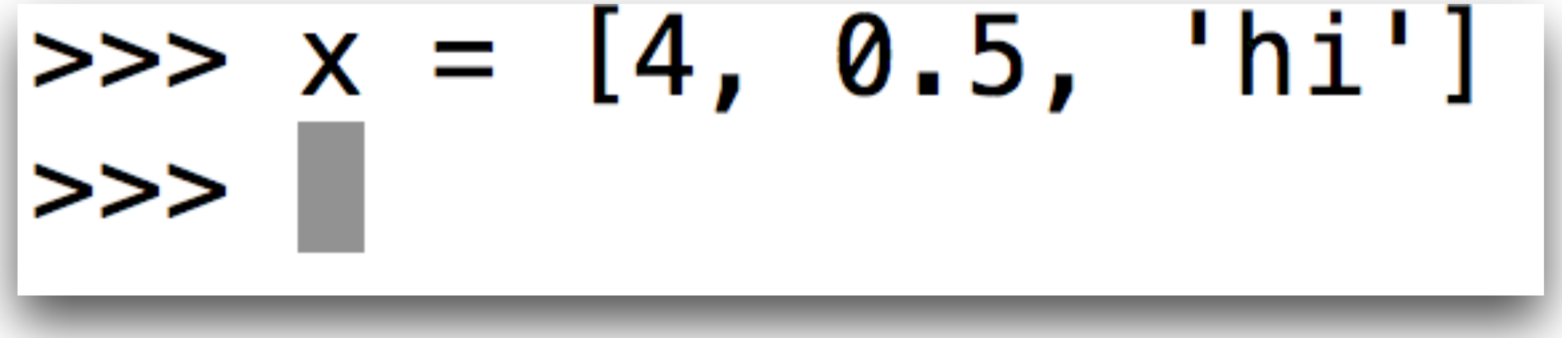
What about Python lists?

- How are Python lists represented internally?
  - Remember: they are arrays.
  - But they are also **objects**.

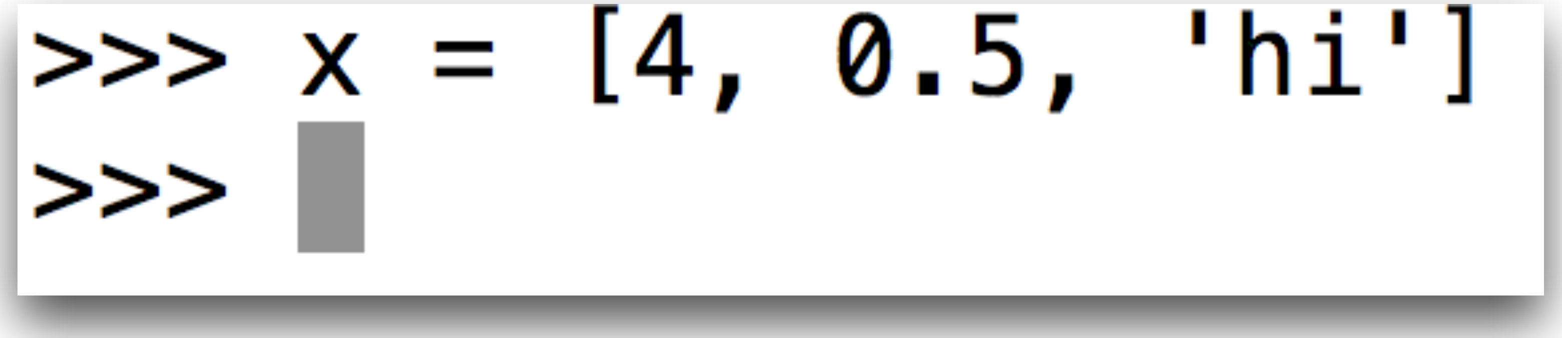


- The **data**
- The **type of the data**
- Other stuff…

The “object”

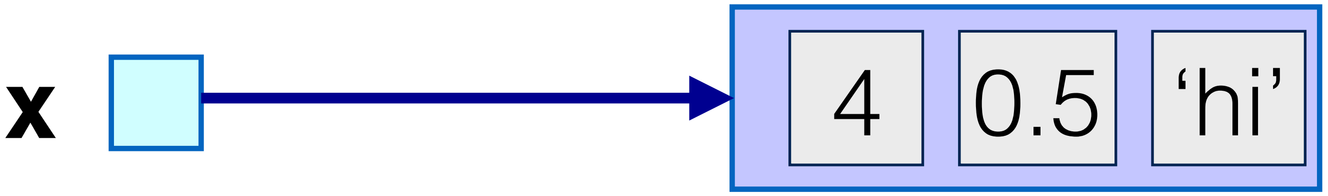


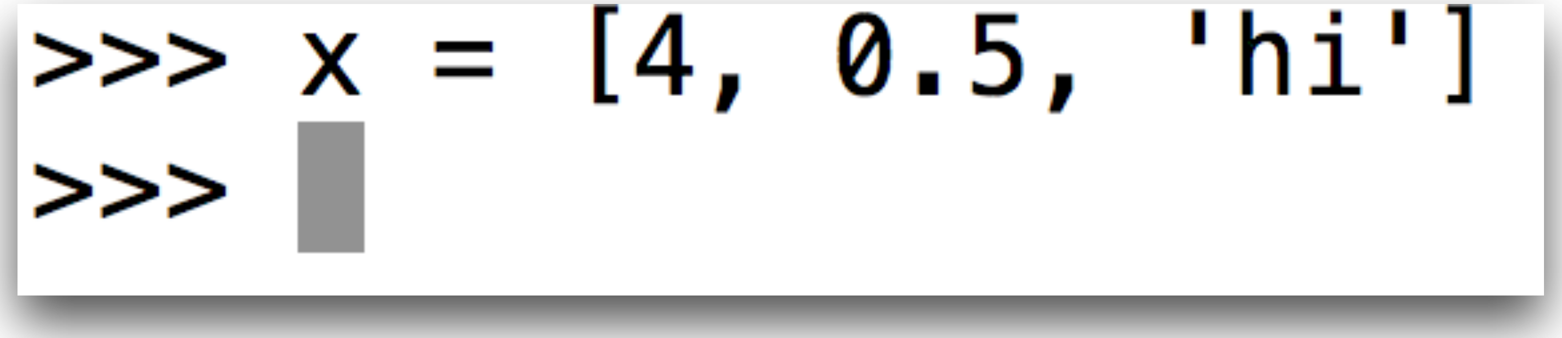
```
>>> x = [4, 0.5, 'hi']
>>> 
```



```
>>> x = [4, 0.5, 'hi']
>>> 
```

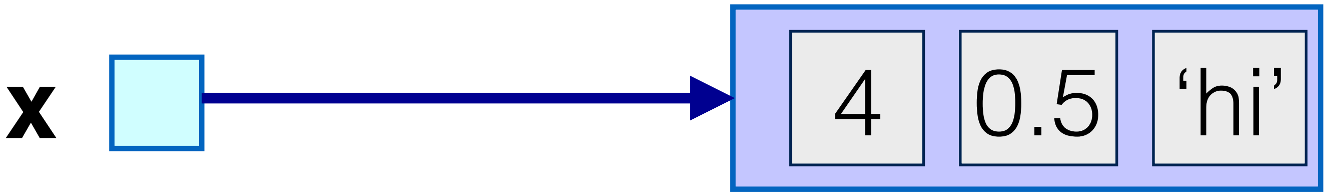
Like this?



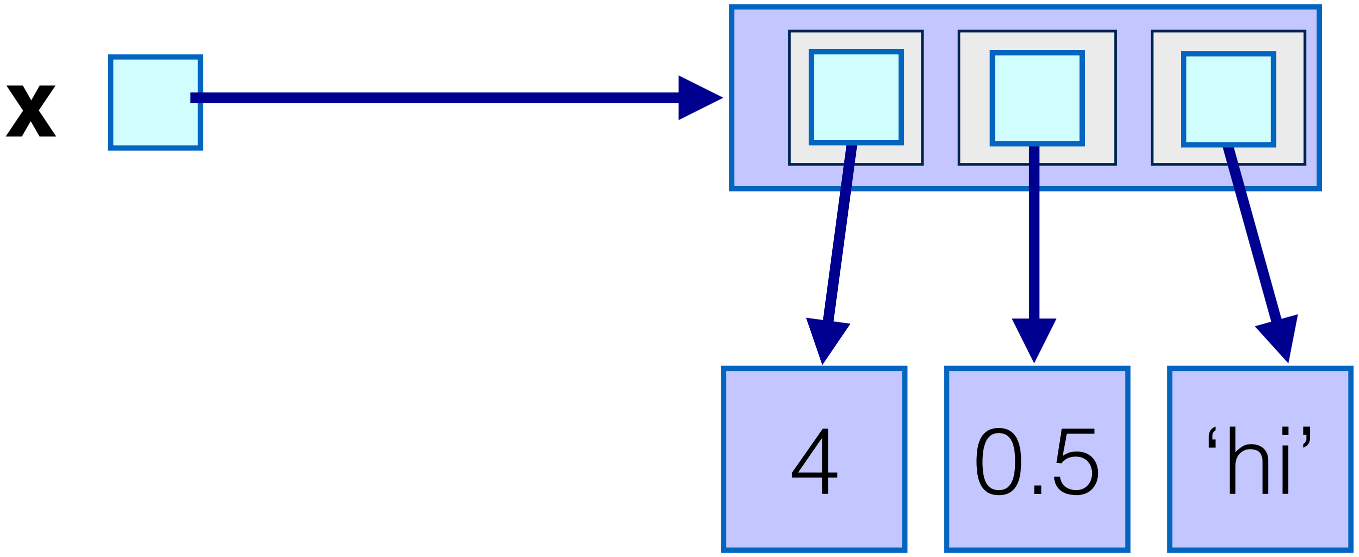


```
>>> x = [4, 0.5, 'hi']
>>> 
```

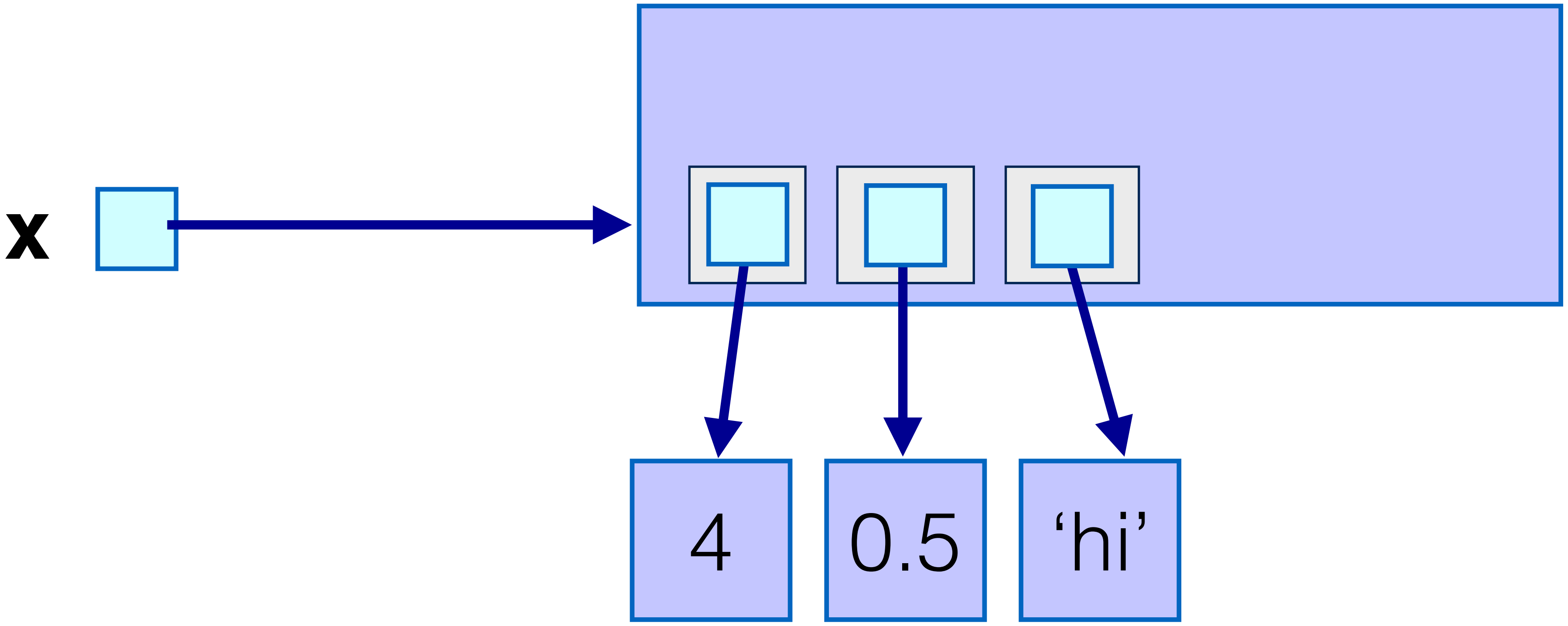
Like this?



Close, but not quite…

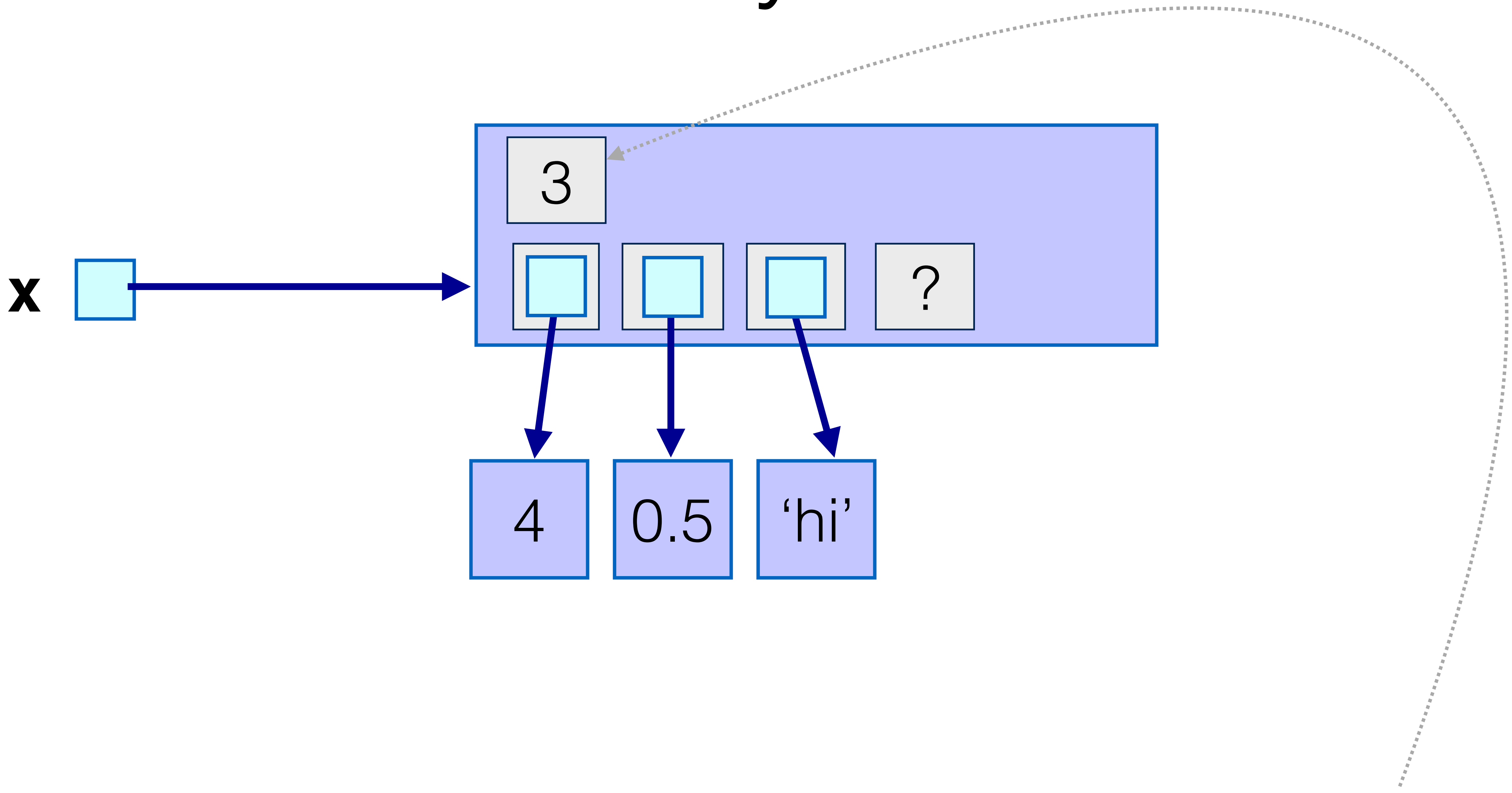


What about Python lists?



- The **object** list also contains other information, i.e., **length**.
- The **key point** is that **they are arrays of references**.

What about Python lists?



- The **object** list also contains other information, i.e., **length**.
- The **key point** is that **they are arrays of references**.

Some objects never change

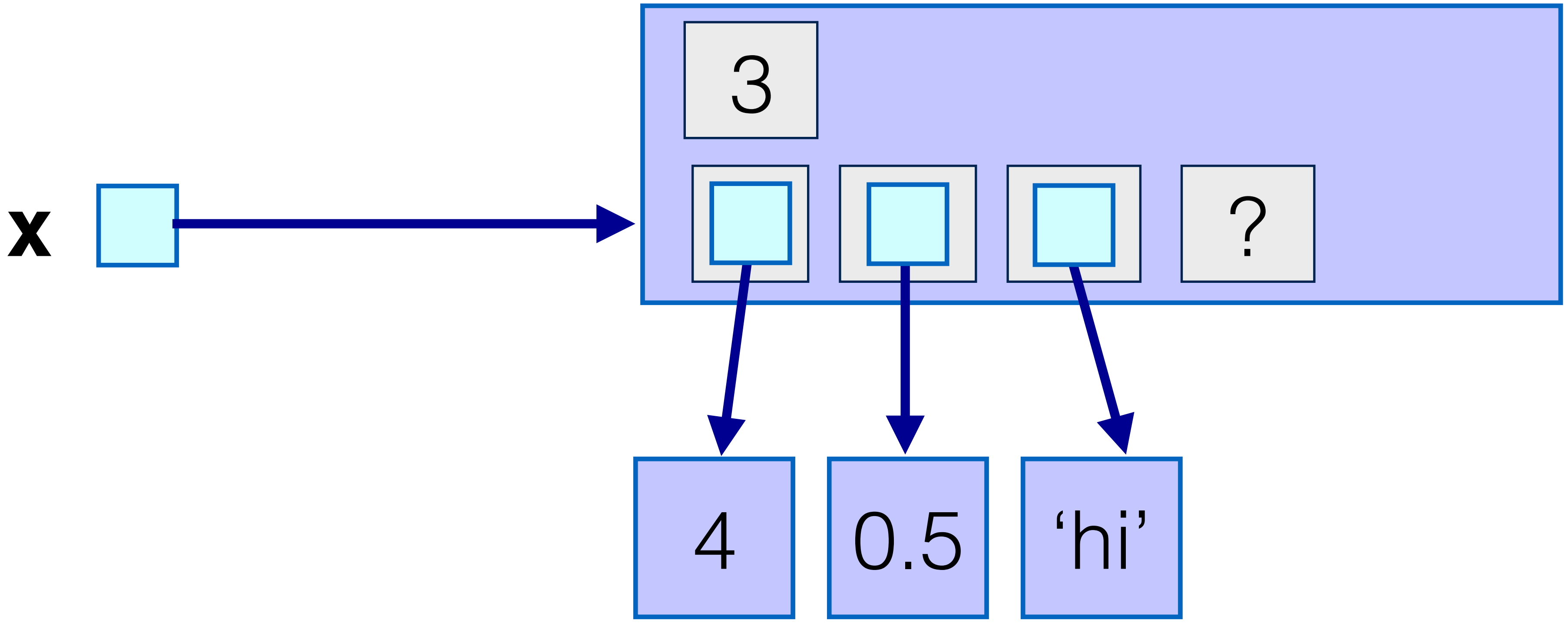
in Python

They are called **immutable**.

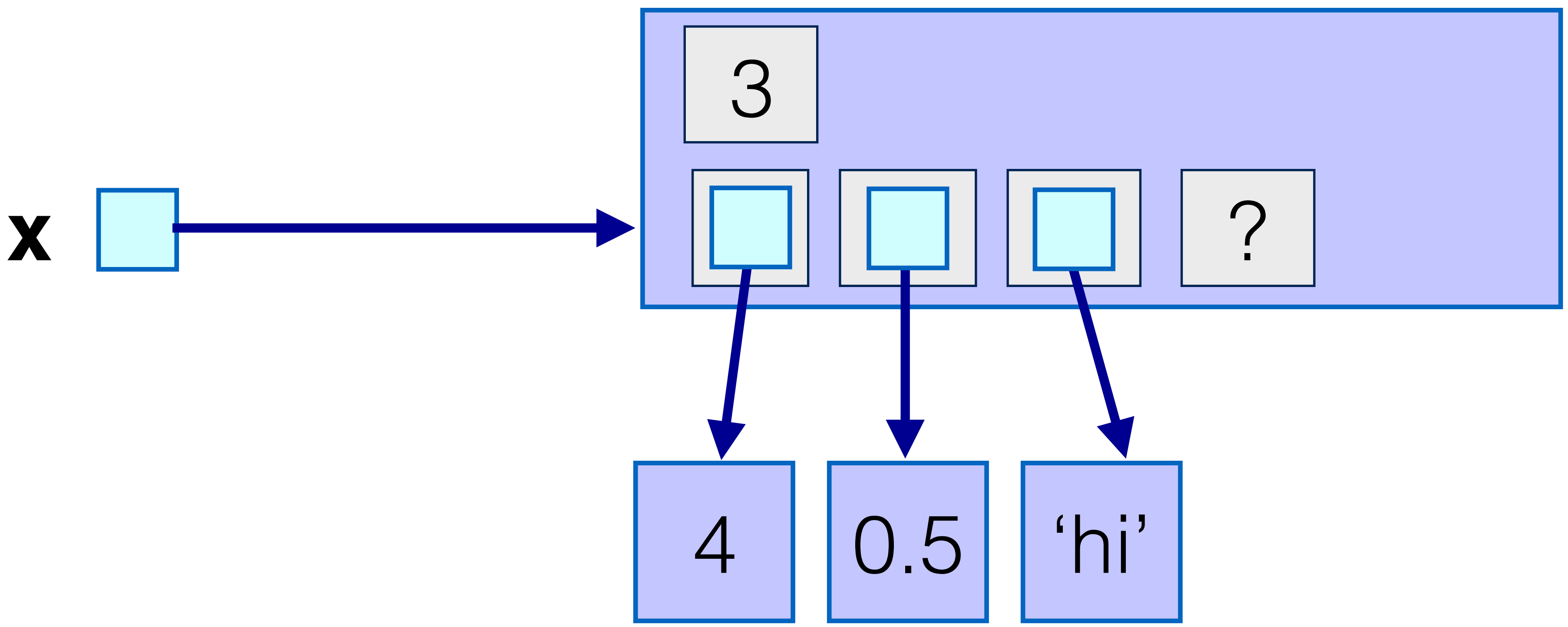
Mutable/Immutable

- Lists are **mutable**:
  - In other words: objects of type list in Python can be changed without creating a new object.

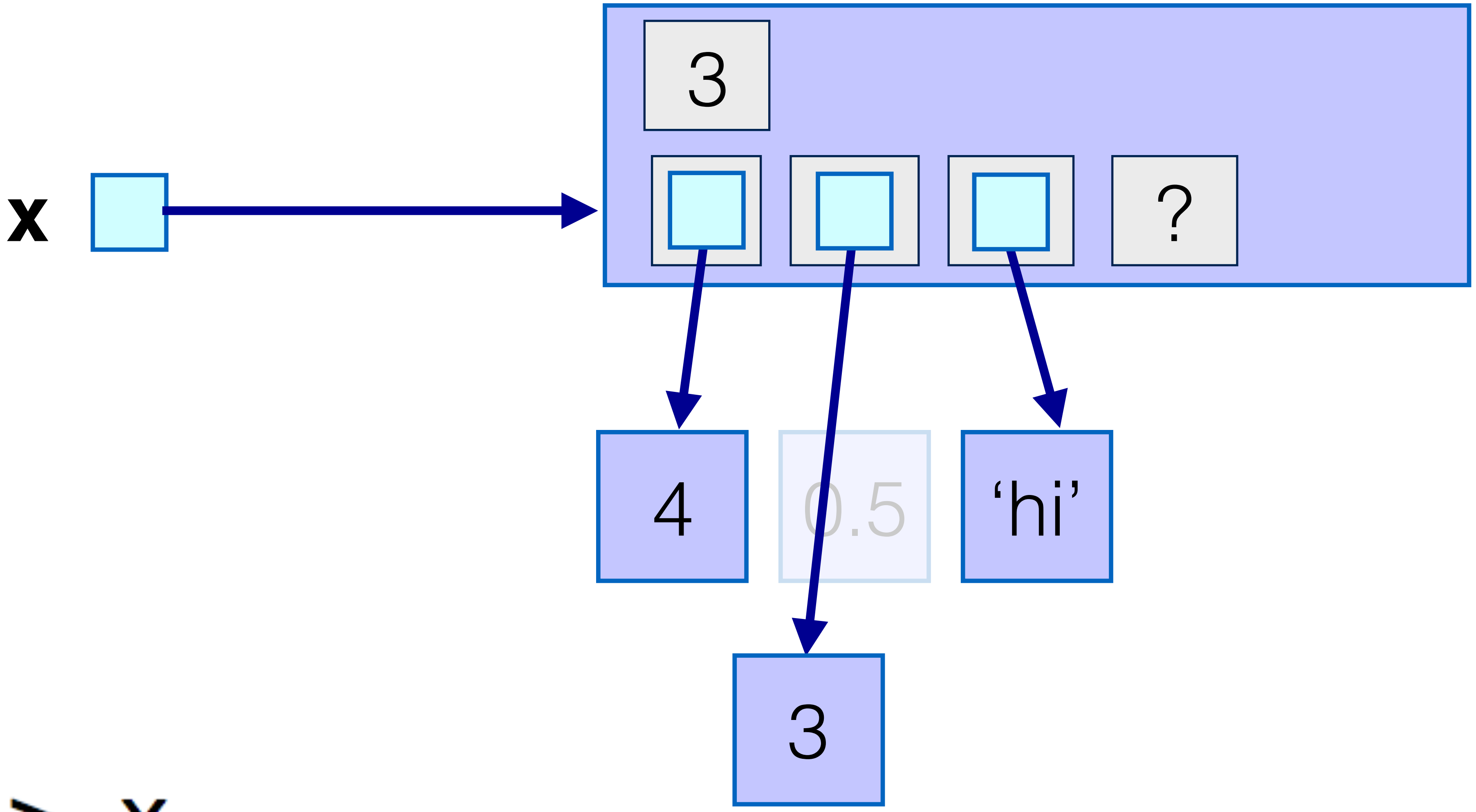
- Integers are **immutable:**
  - Once created they **cannot be changed**
  - I can create a new one, but not modify an already created one.



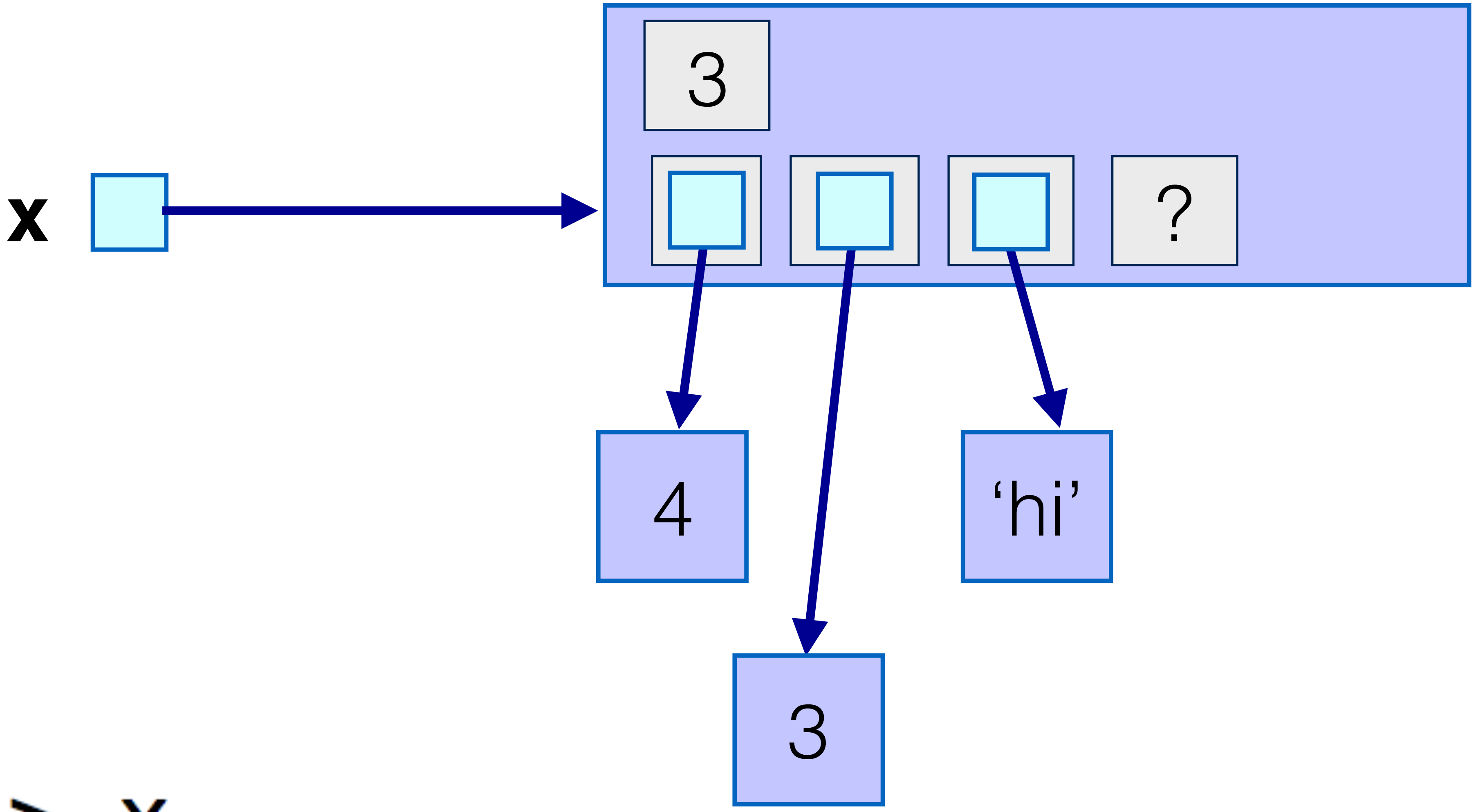
```
>>> x
[4, 0.5, 'hi']
>>> x[1] = 3
>>> x
[4, 3, 'hi']
```

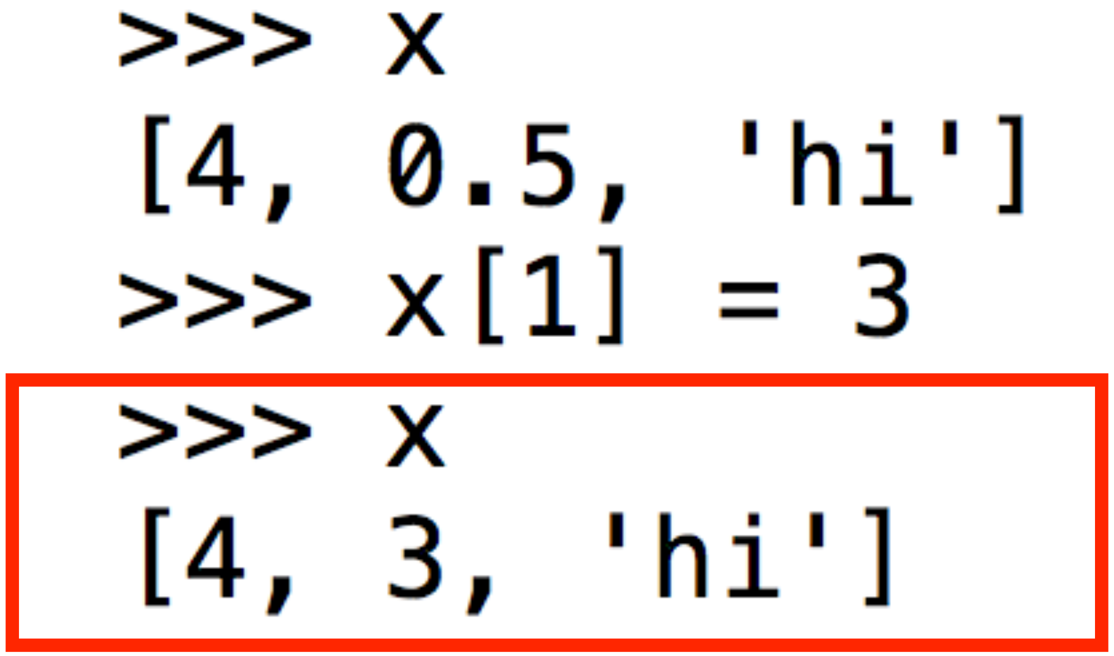


```
>>> x
[4, 0.5, 'hi']
>>> x[1] = 3
>>> x
[4, 3, 'hi']
```



```
>>> x
[4, 0.5, 'hi']
>>> x[1] = 3
>>> x
[4, 3, 'hi']
```





```
>>> x
[4, 0.5, 'hi']
>>> x[1] = 3
>>> x
[4, 3, 'hi']
```

I am changing the **list** object!

Note that a new **integer** object has been created.

- List are **mutable**:
  - In other words: objects of type list in Python can be changed.

- Integers are **immutable:**
  - Once created they **cannot be changed**
  - I can create a new one, but not modify an already created one.
- Strings are **immutable**.

Hello
my name is

Names

- First remember, in Python **all identifiers are names**: variables, functions, methods, modules, types, …

- This means, **a name can only refer to one thing** at a time!

- Careful when reusing names then…

Example

```
>>> a_name = 10*6
>>> a_name
60
```

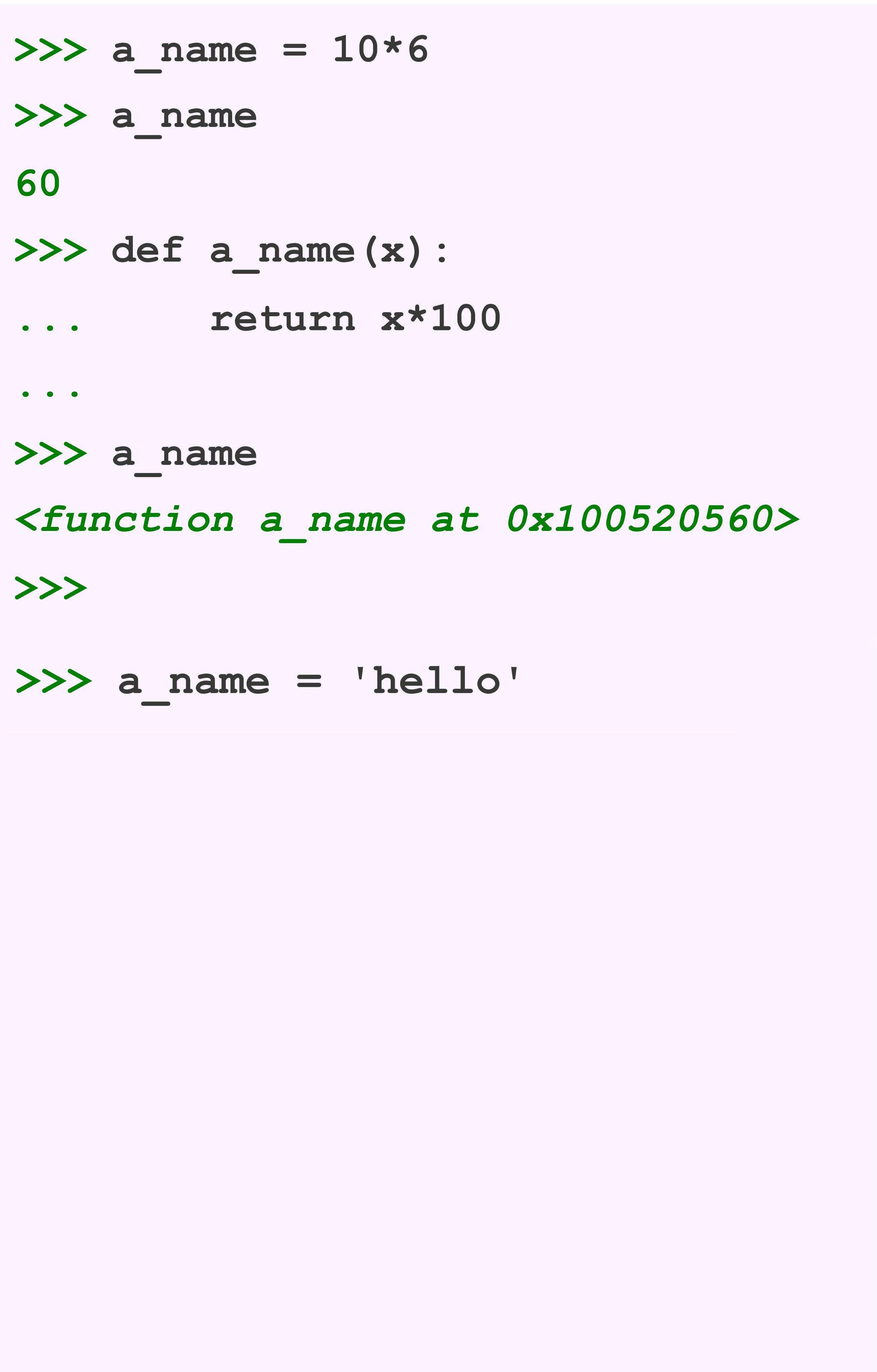
Example

```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
```

Example

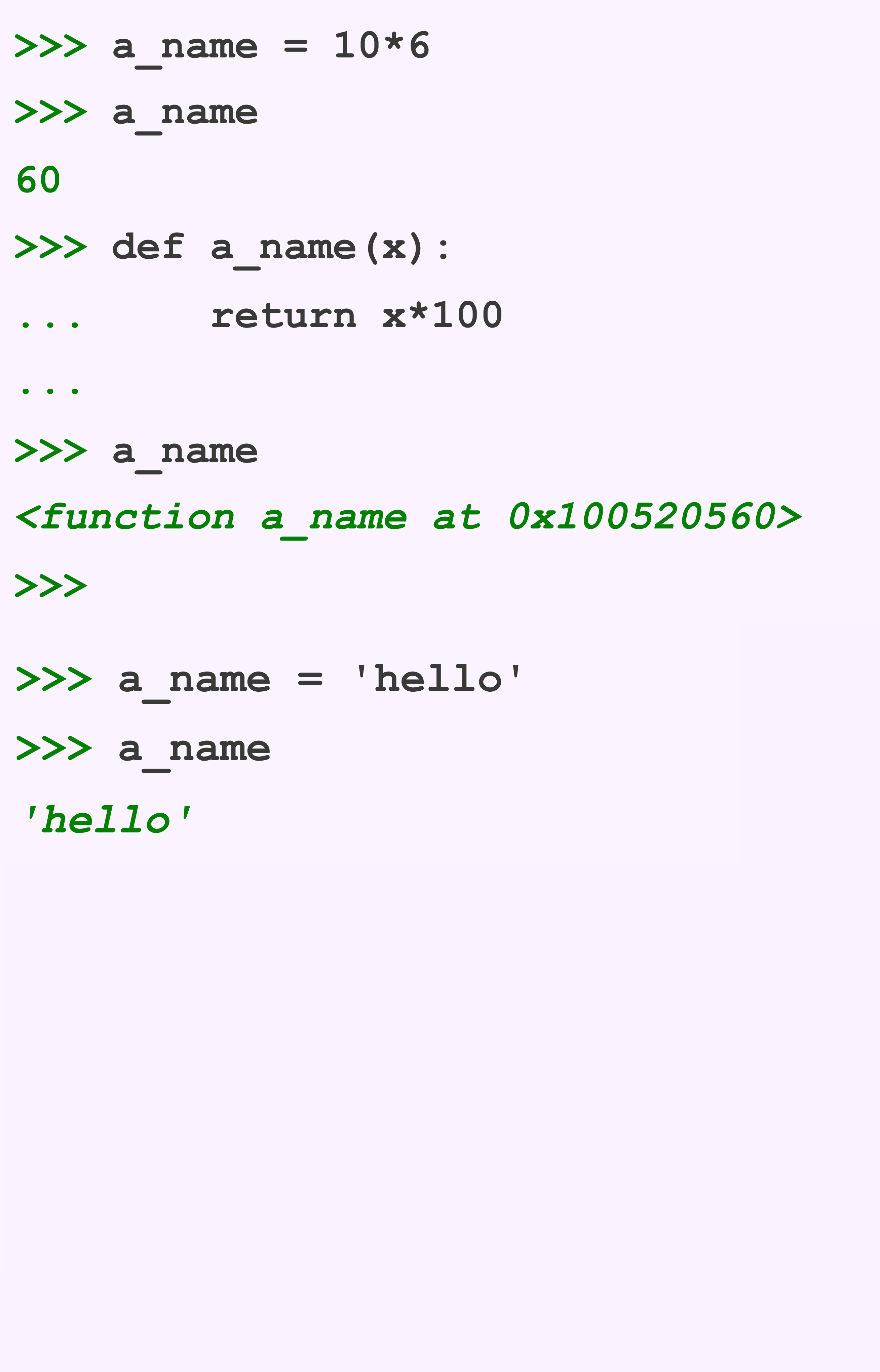
```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
>>> a_name
<function a_name at 0x100520560>
```

Example



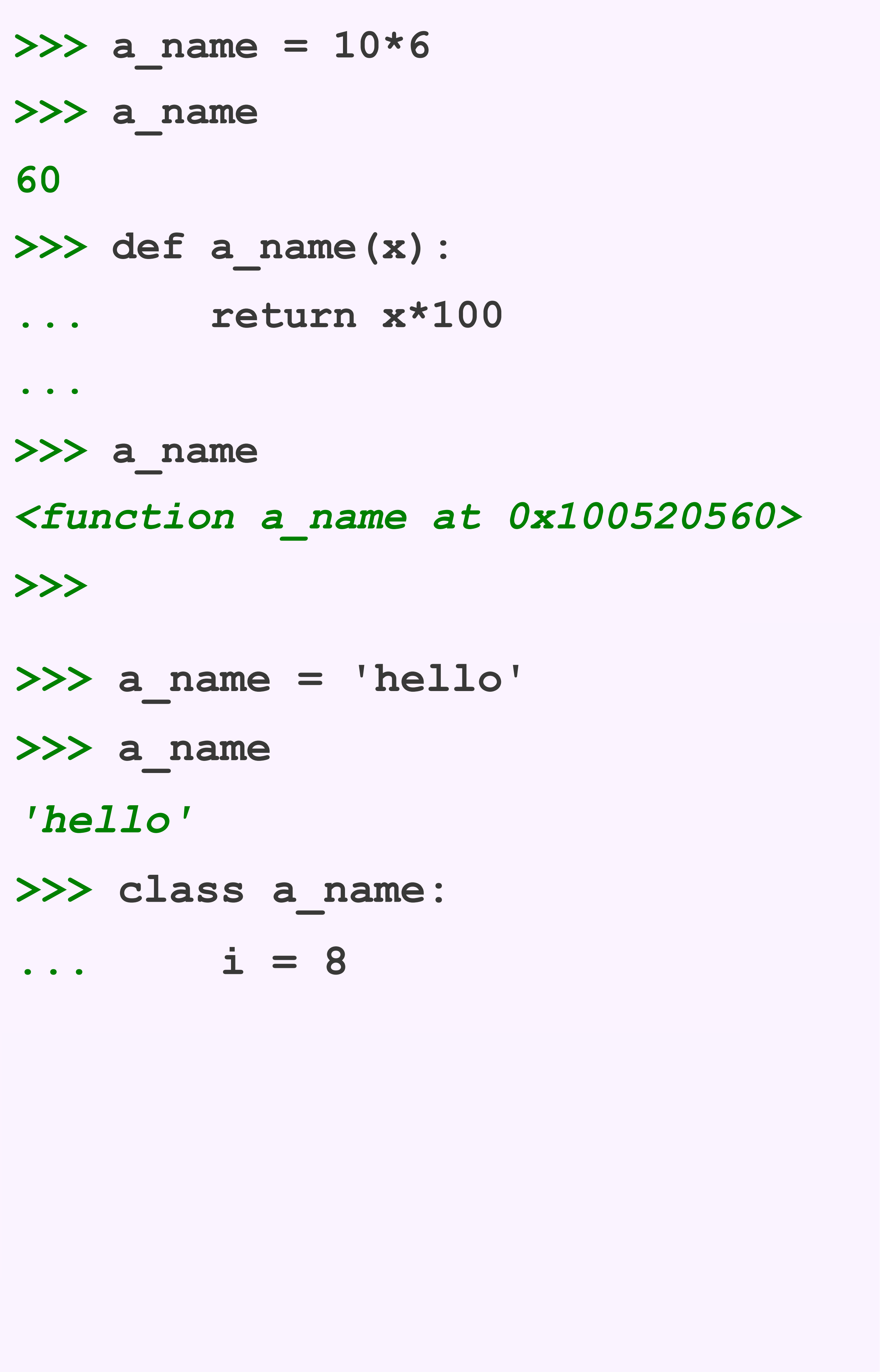
```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
>>> a_name
<function a_name at 0x100520560>
>>>

>>> a_name = 'hello'
```

Example



```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
>>> a_name
<function a_name at 0x100520560>
>>>

>>> a_name = 'hello'
>>> a_name
'hello'
```

Example



```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
>>> a_name
<function a_name at 0x100520560>
>>>

>>> a_name = 'hello'
>>> a_name
'hello'
>>> class a_name:
...     i = 8
```

Example

```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
>>> a_name
<function a_name at 0x100520560>
>>>

>>> a_name = 'hello'
>>> a_name
'hello'
>>> class a_name:
...     i = 8
...
>>> a_name
<class '__main__.a_name'>
```

Example

```
>>> a_name = 10*6
>>> a_name
60
>>> def a_name(x):
...     return x*100
...
>>> a_name
<function a_name at 0x100520560>
>>>

>>> a_name = 'hello'
>>> a_name
'hello'
>>> class a_name:
...     i = 8
...
>>> a_name
<class '__main__.a_name'>
```

Single variable…
one name for different objects

Namespaces (or environments)

Namespaces (or environments)

- A **namespace** is a mapping of names to objects: like a dictionary

Namespaces (or environments)

- A **namespace** is a mapping of names to objects: like a dictionary
- When the interpreter starts, it creates a namespace with the names of the built-in functions

Namespaces (or environments)

- A **namespace** is a mapping of names to objects: like a dictionary
- When the interpreter starts, it creates a namespace with the names of the built-in functions
- Each file (also called **module**) has its own namespace.
  - Don’t put two classes or two functions with the same name in a file
  - They share the same namespace, so the result can be surprising. With two functions, the second definition overwrites the first.

Namespaces (or environments)

- A **namespace** is a mapping of names to objects: like a dictionary
- When the interpreter starts, it creates a namespace with the names of the built-in functions
- Each file (also called **module**) has its own namespace.
  - Don’t put two classes or two functions with the same name in a file
  - They share the same namespace, so the result can be surprising. With two functions, the second definition overwrites the first.
- **Functions** have their namespace too. When a function is called, Python creates a local namespace for it. This namespace is forgotten once the function finishes.

Namespaces (or environments)

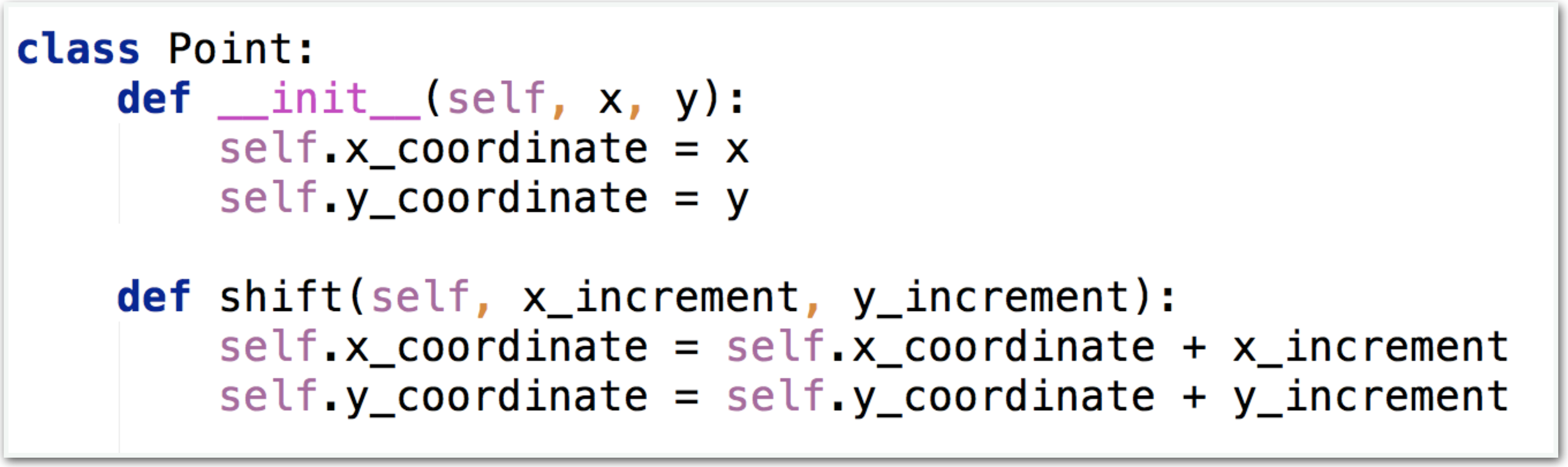
- A **namespace** is a mapping of names to objects: like a dictionary

- When the interpreter starts, it creates a namespace with the names of the built-in functions

- Each file (also called **module**) has its own namespace.
  - Don't put two classes or two functions with the same name in a file
  - They share the same namespace, so the result can be surprising. With two functions, the second definition overwrites the first.

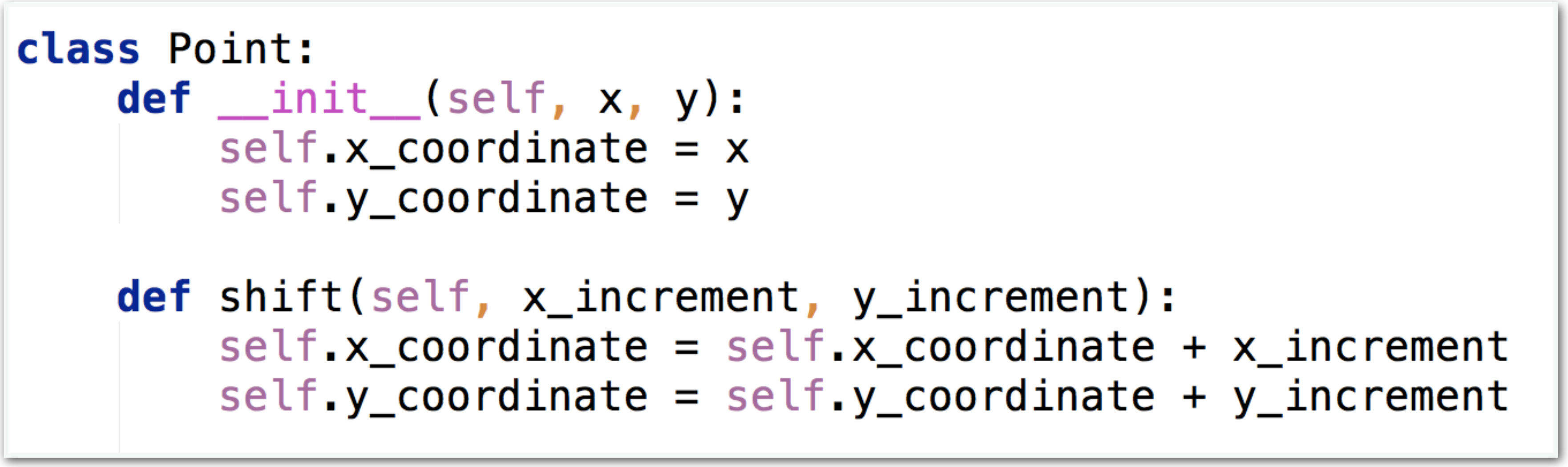
- **Functions** have their namespace too. When a function is called, Python creates a local namespace for it. This namespace is forgotten once the function finishes.

- **Names** belong to the namespace in which they are bound.



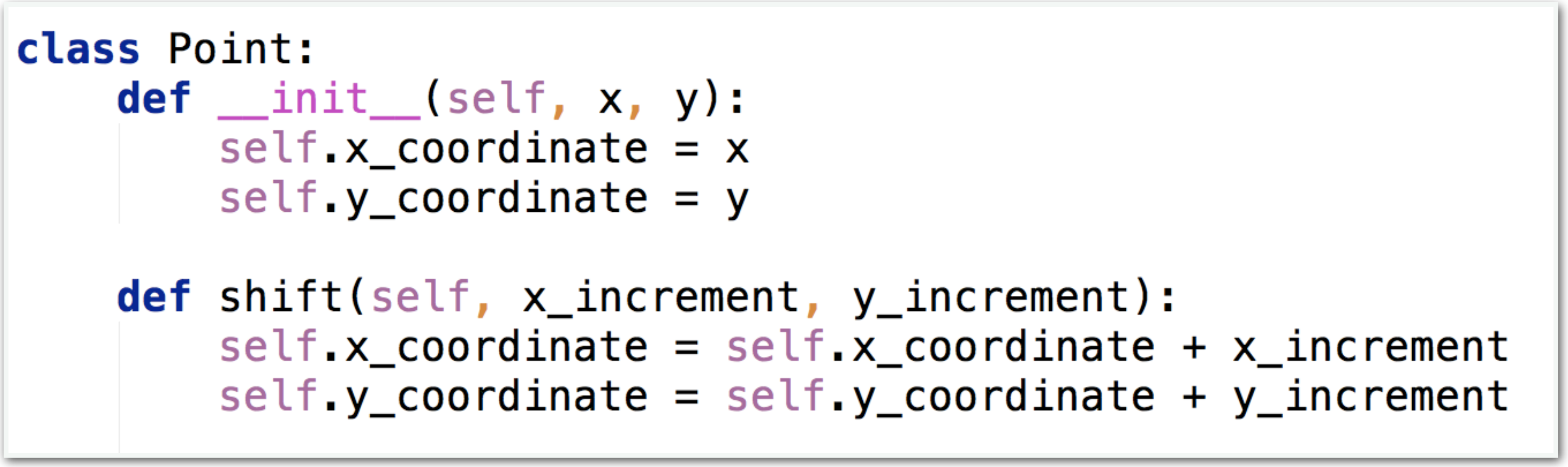
```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```

```
>>> import point
```

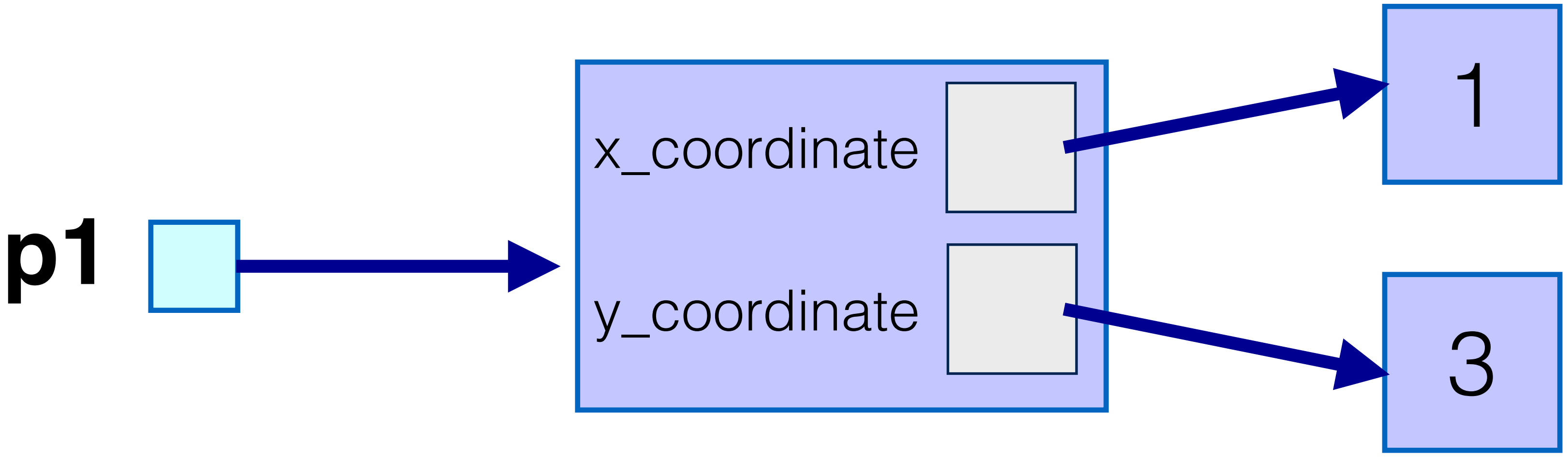


```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```

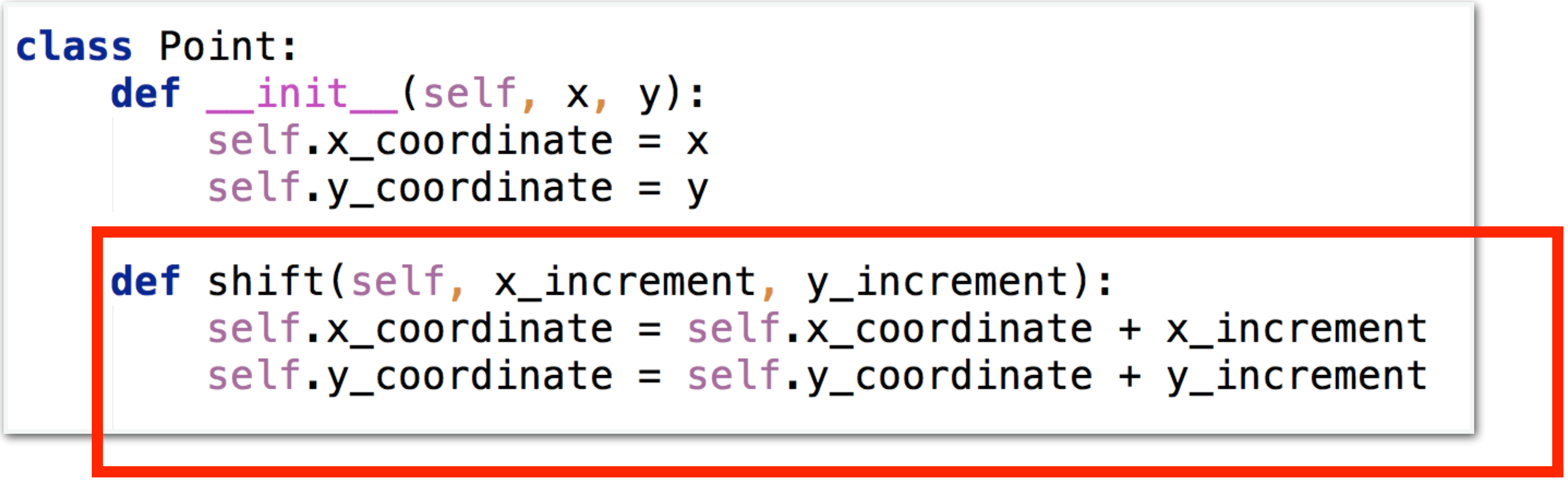
```
>>> import point
>>> p1 = point.Point(1,3)
```



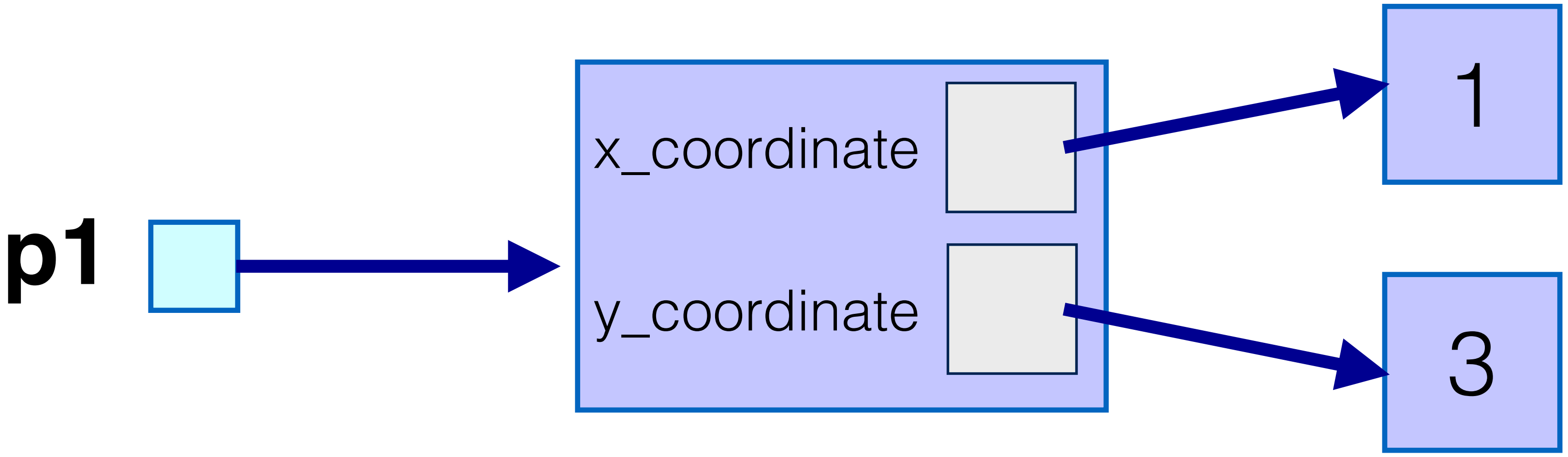
```
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```



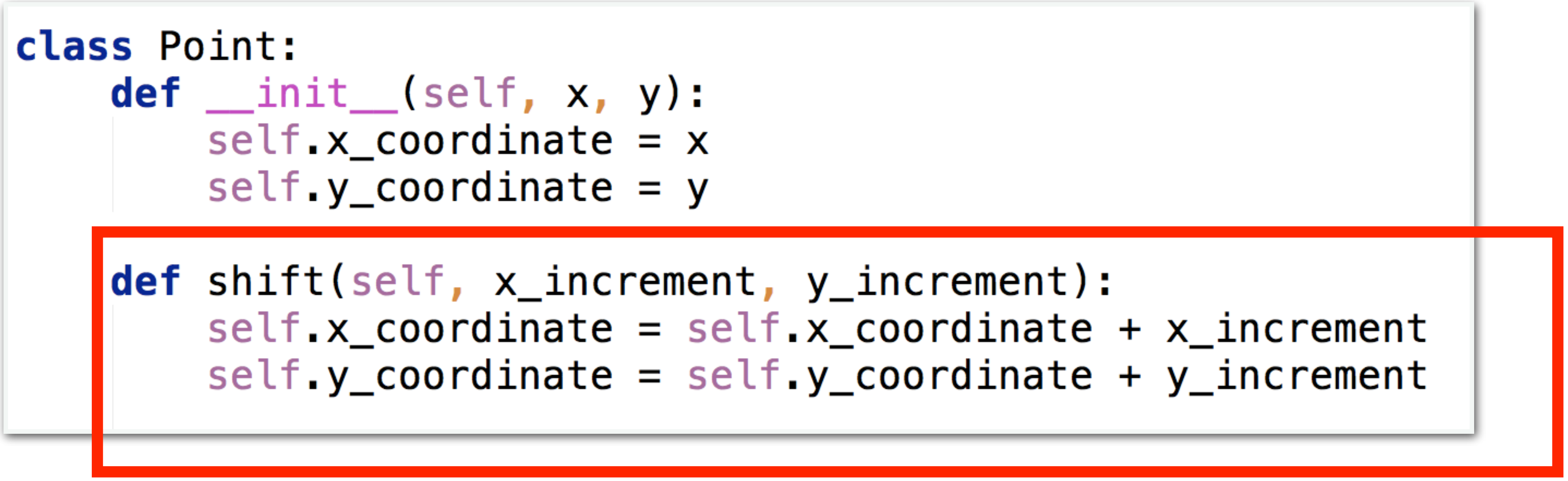
```
>>> import point
>>> p1 = point.Point(1,3)
```



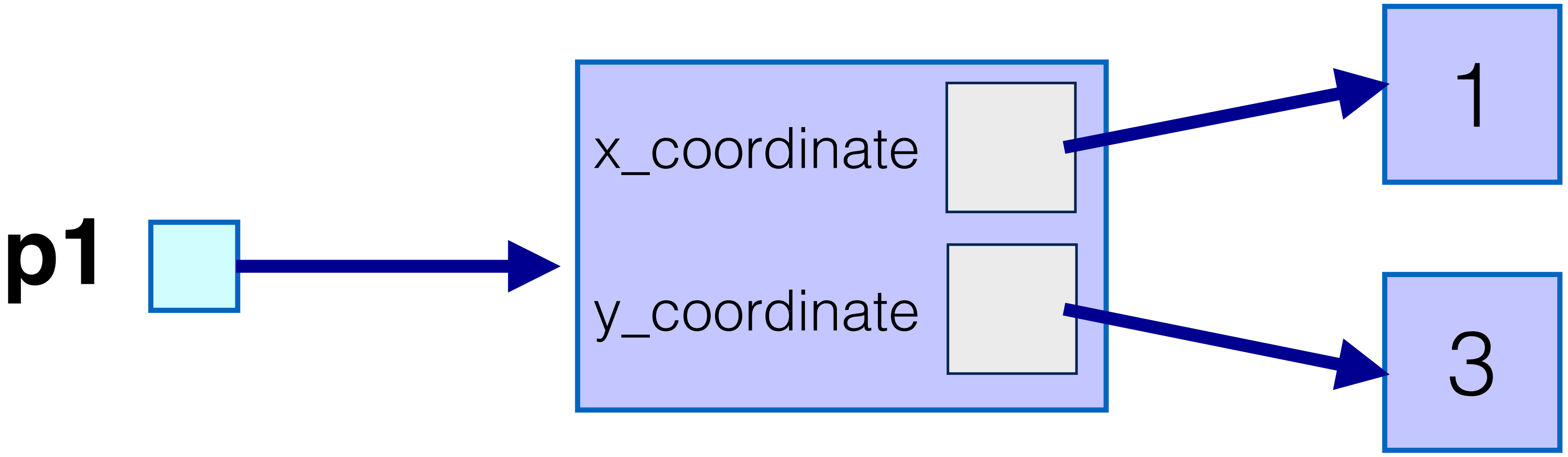
```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```



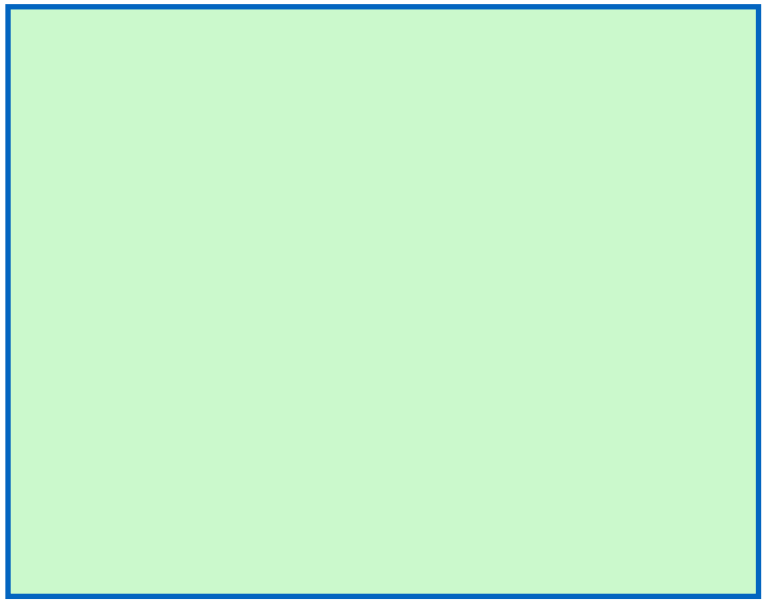
```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.shift(10,20)
```



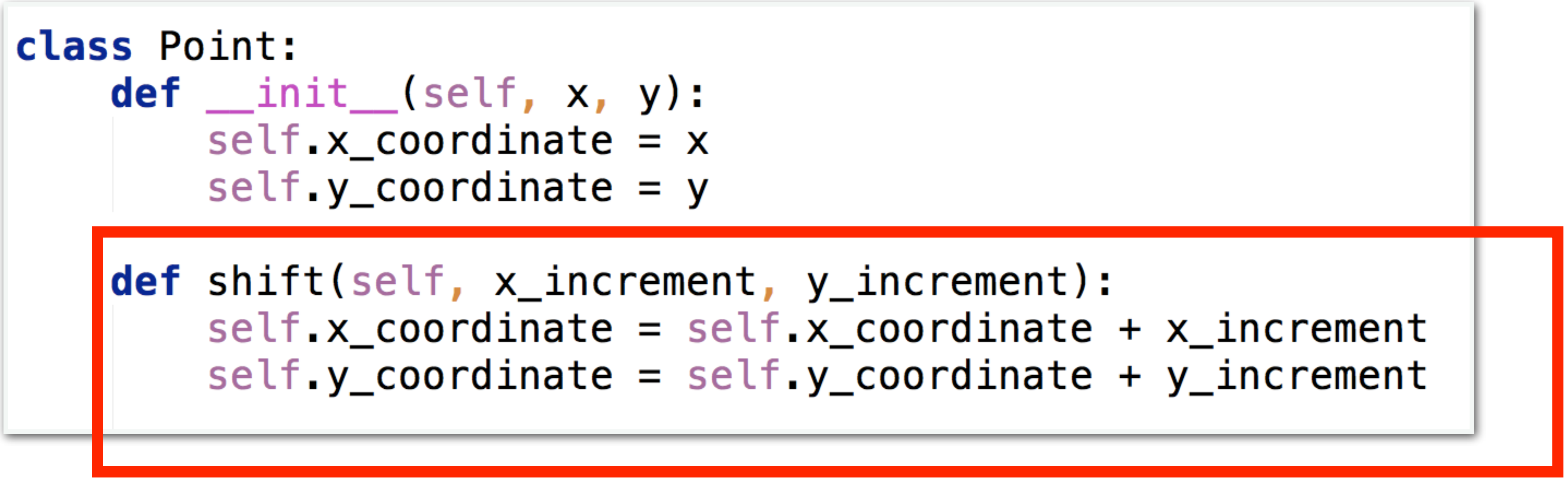
```
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```



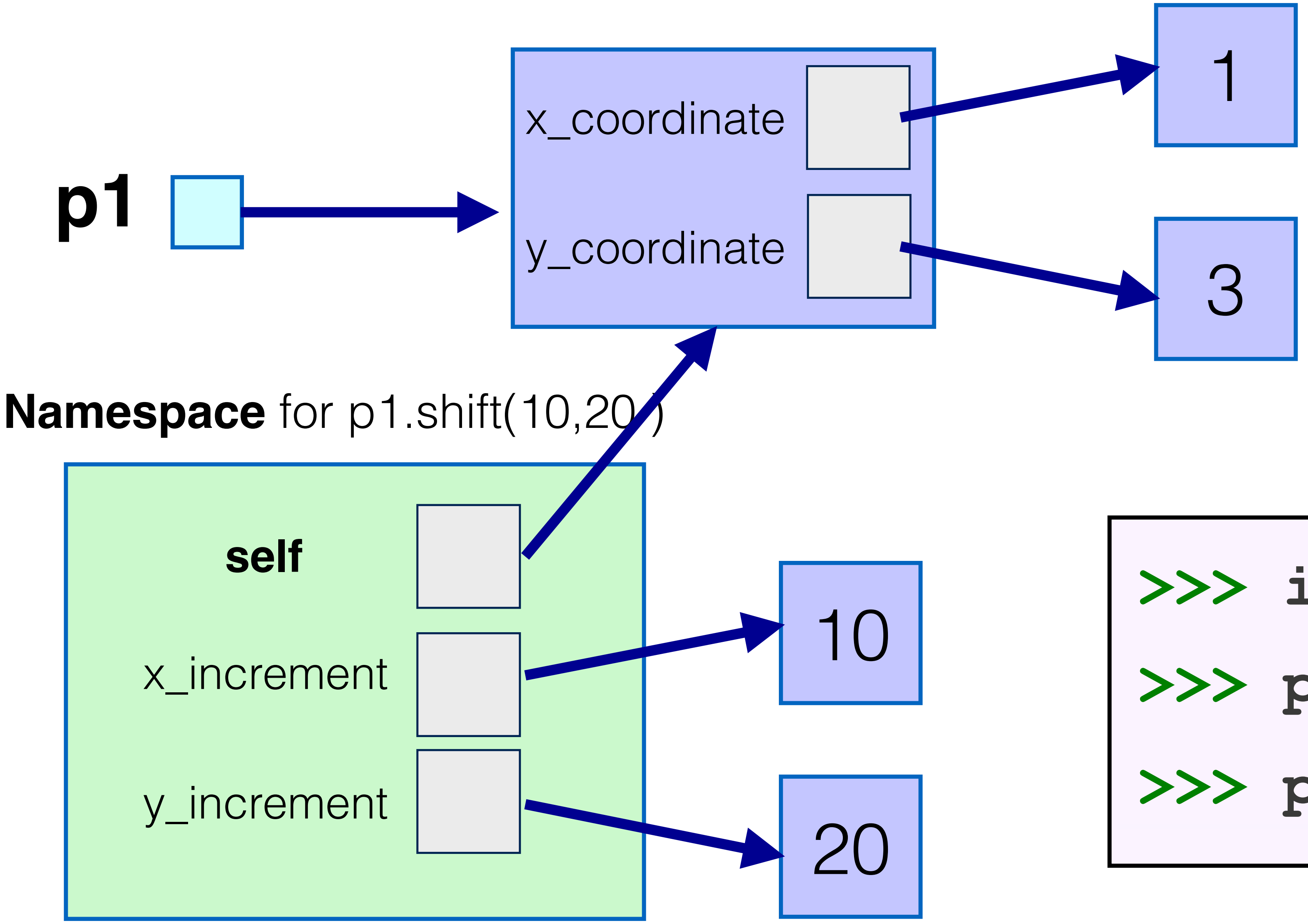
Namespace for p1.shift(10,20)



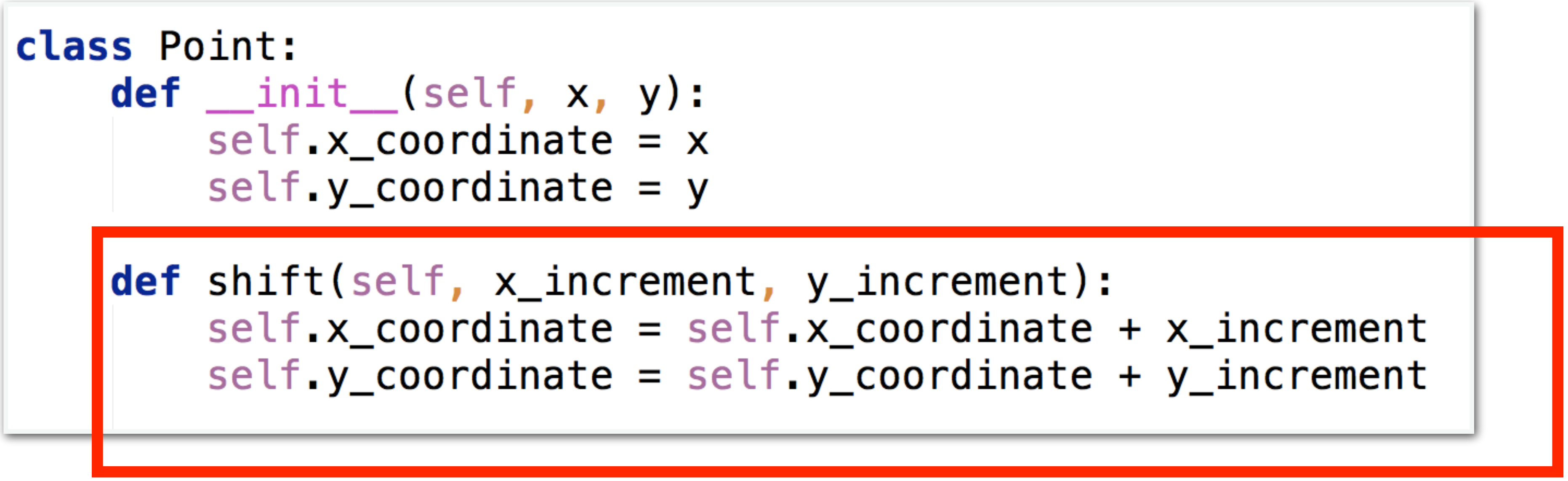
```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.shift(10,20)
```



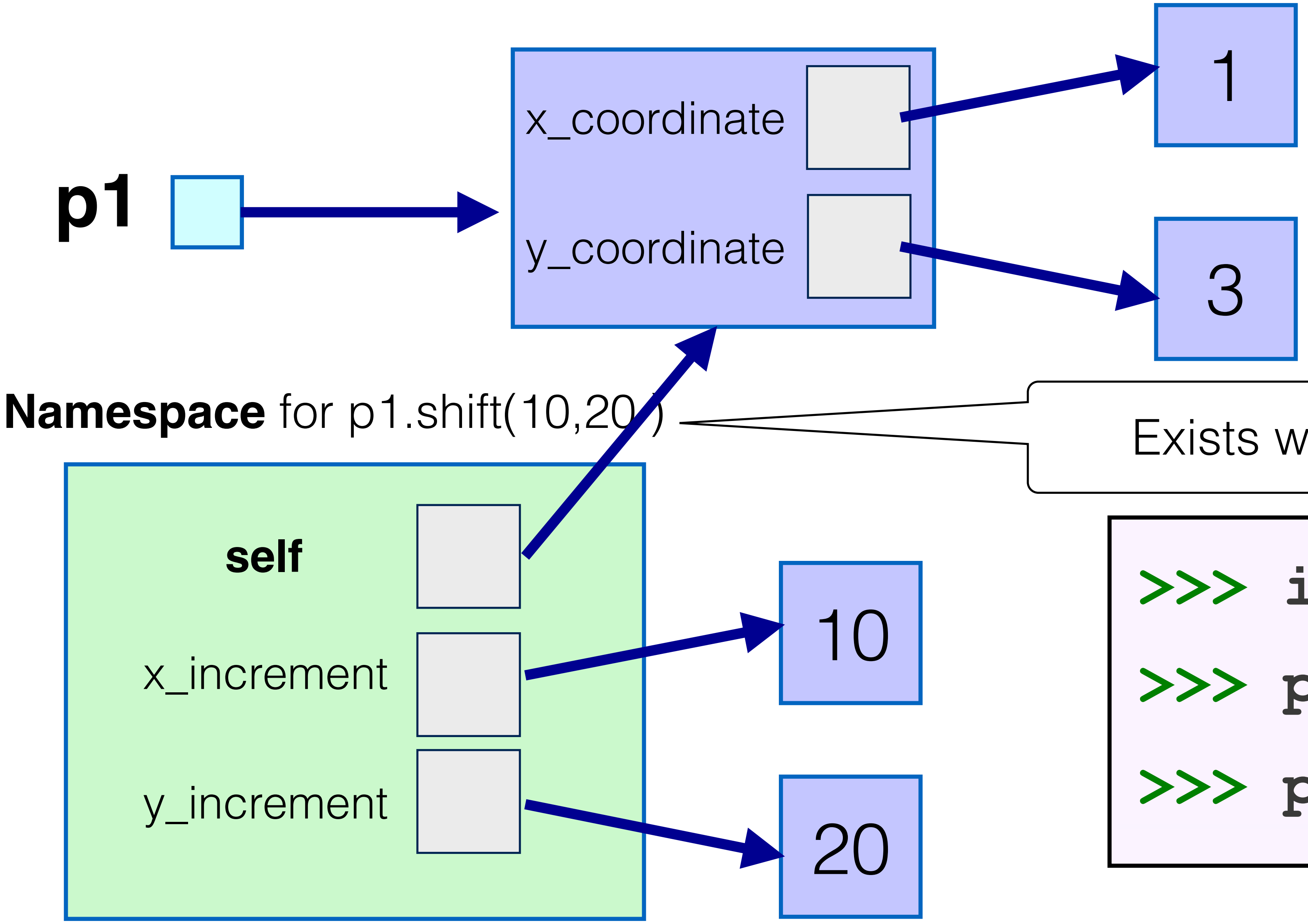
```
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```



```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.shift(10,20)
```

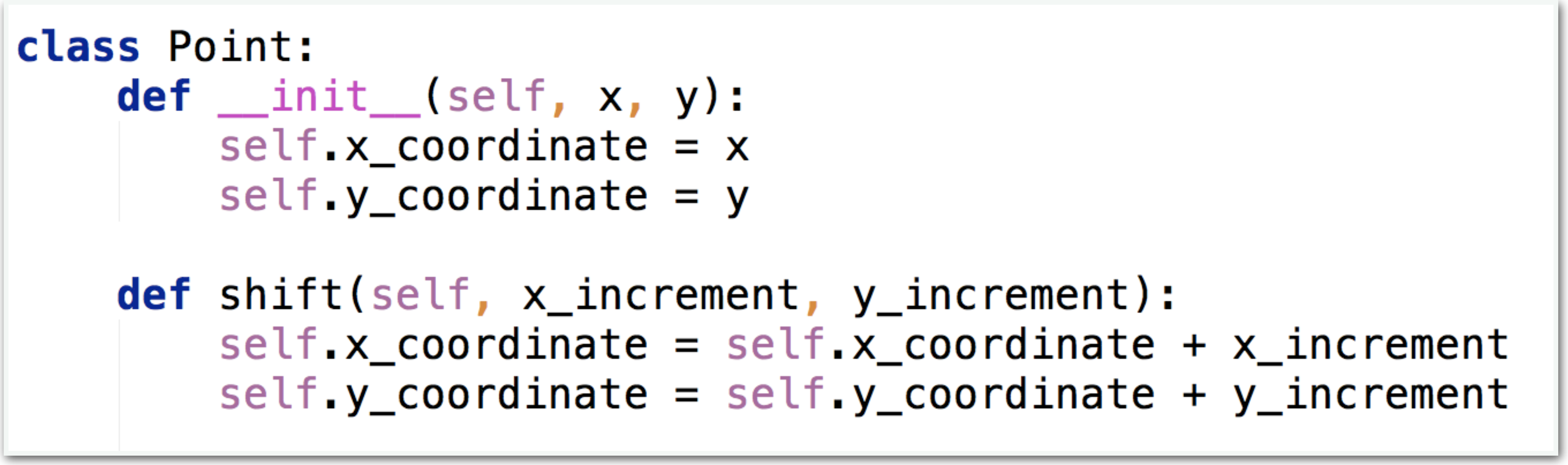


```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```

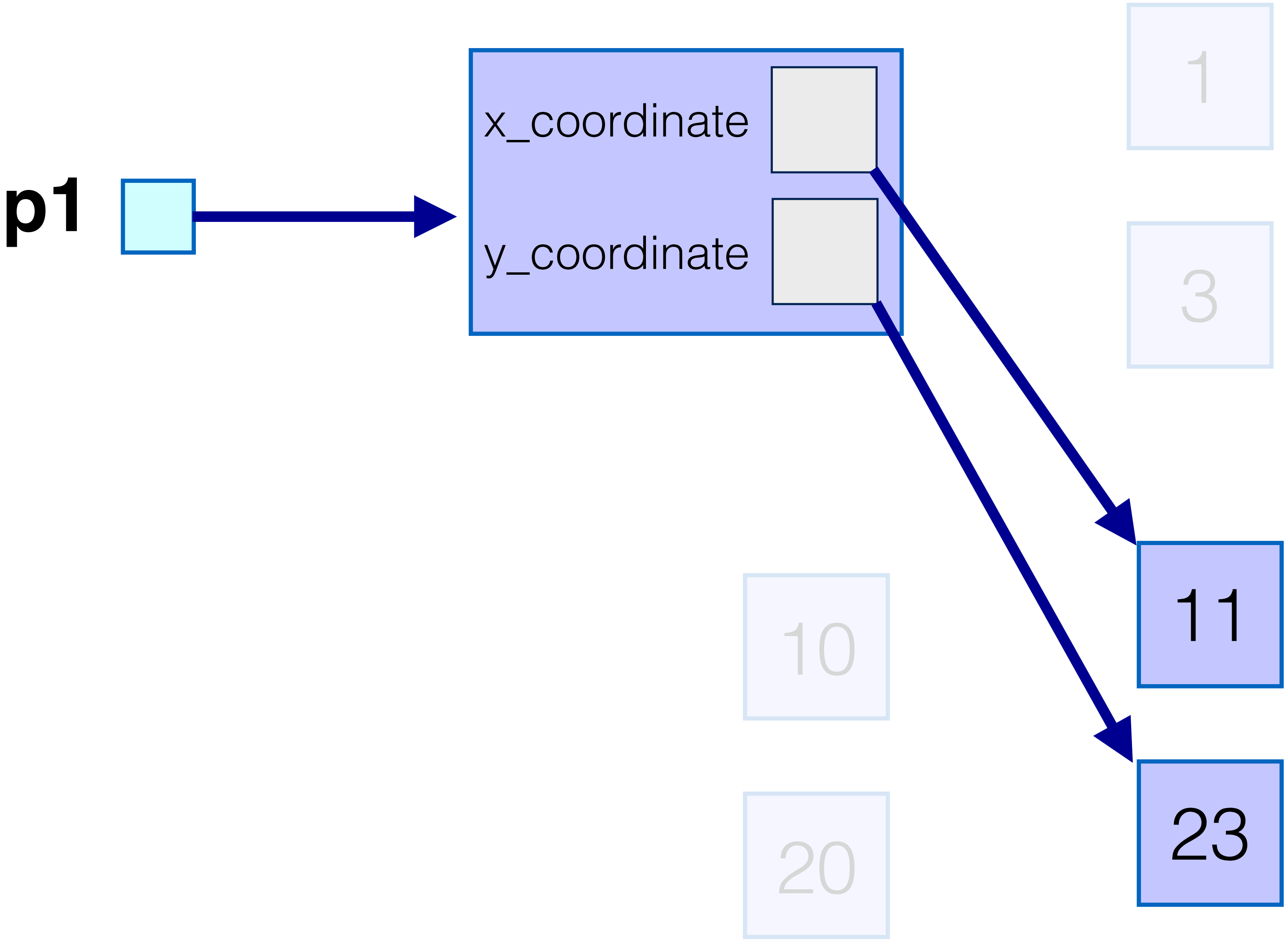


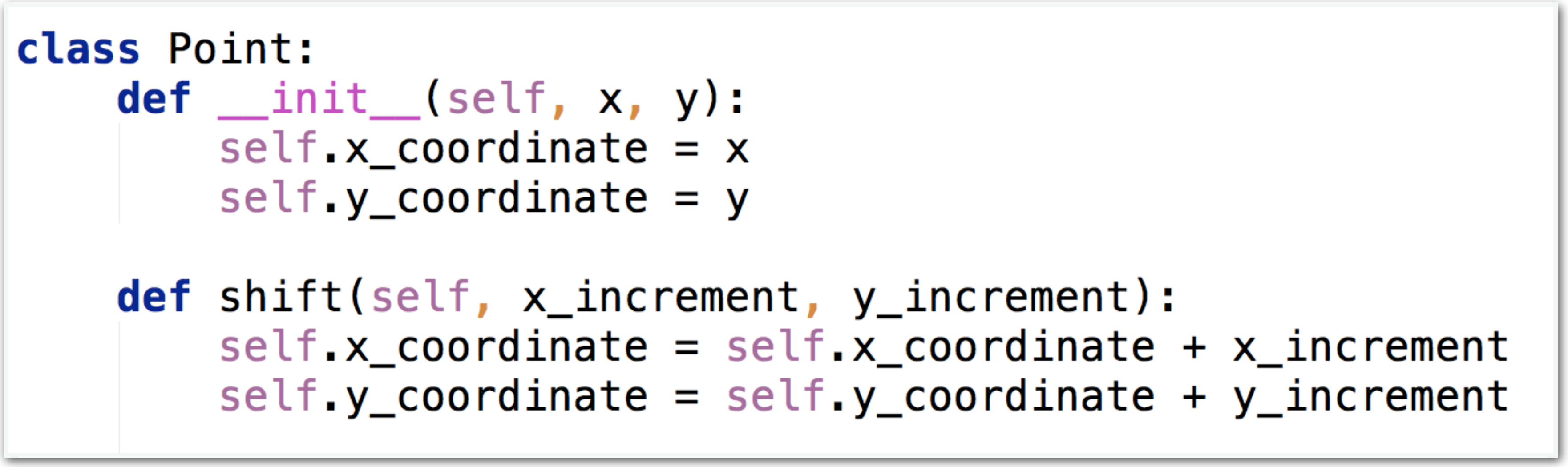
Exists while the function is executing

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.shift(10,20)
```

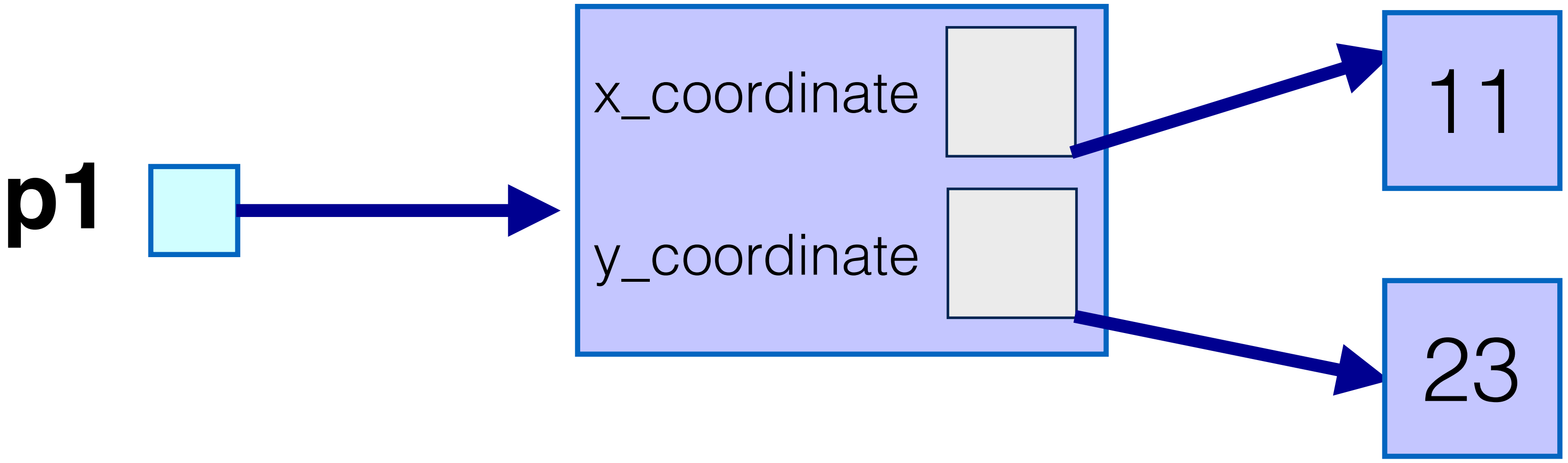


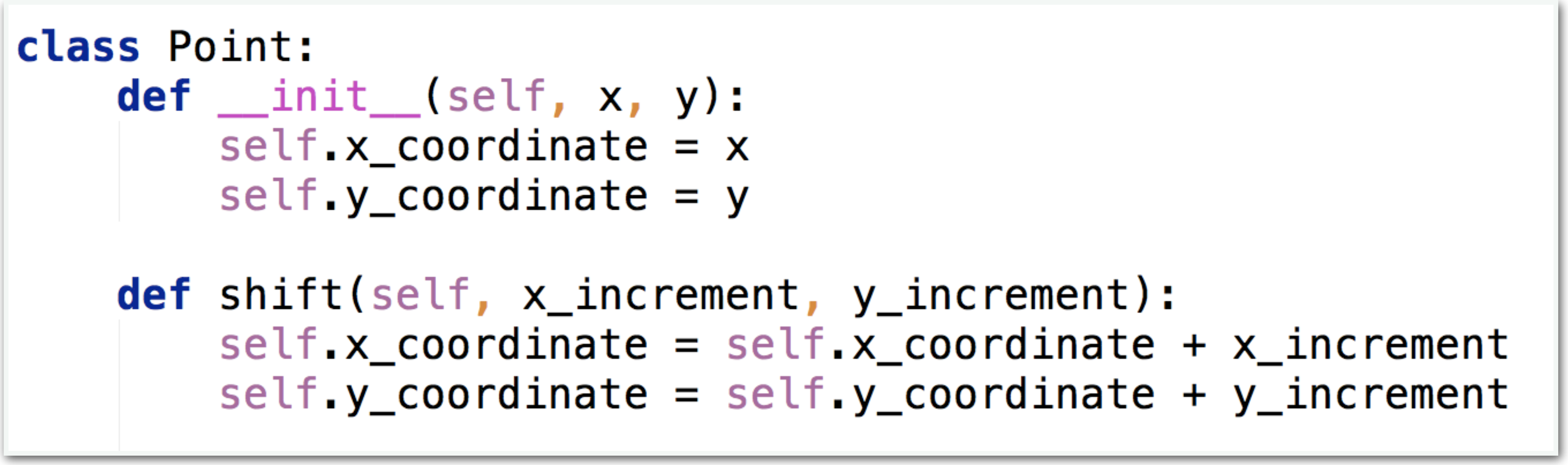
```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```



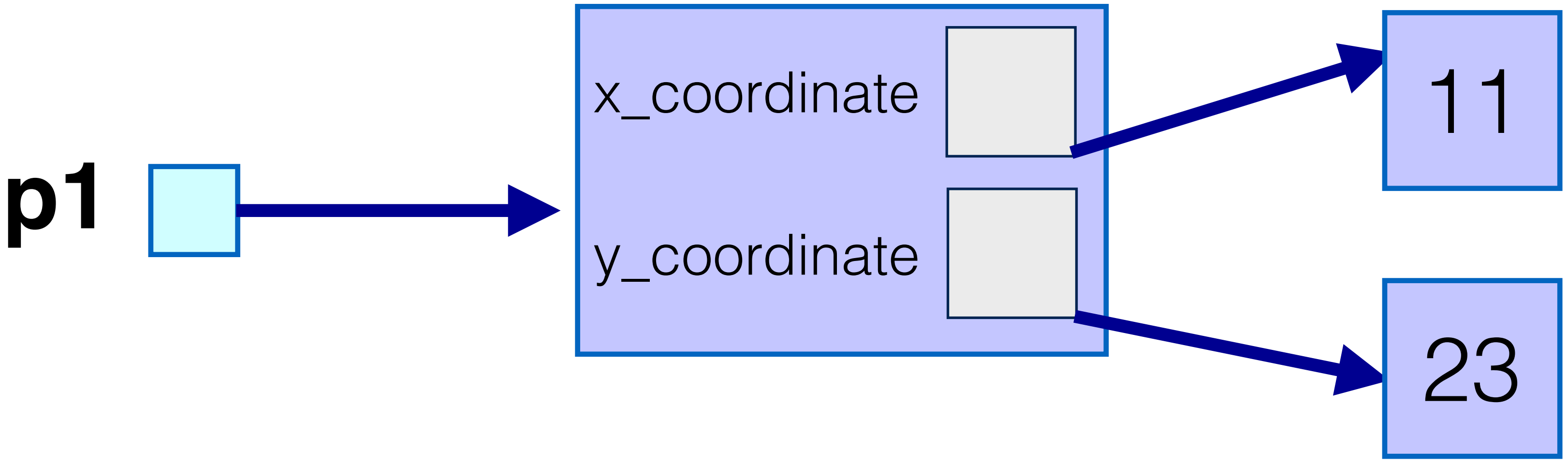


```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```

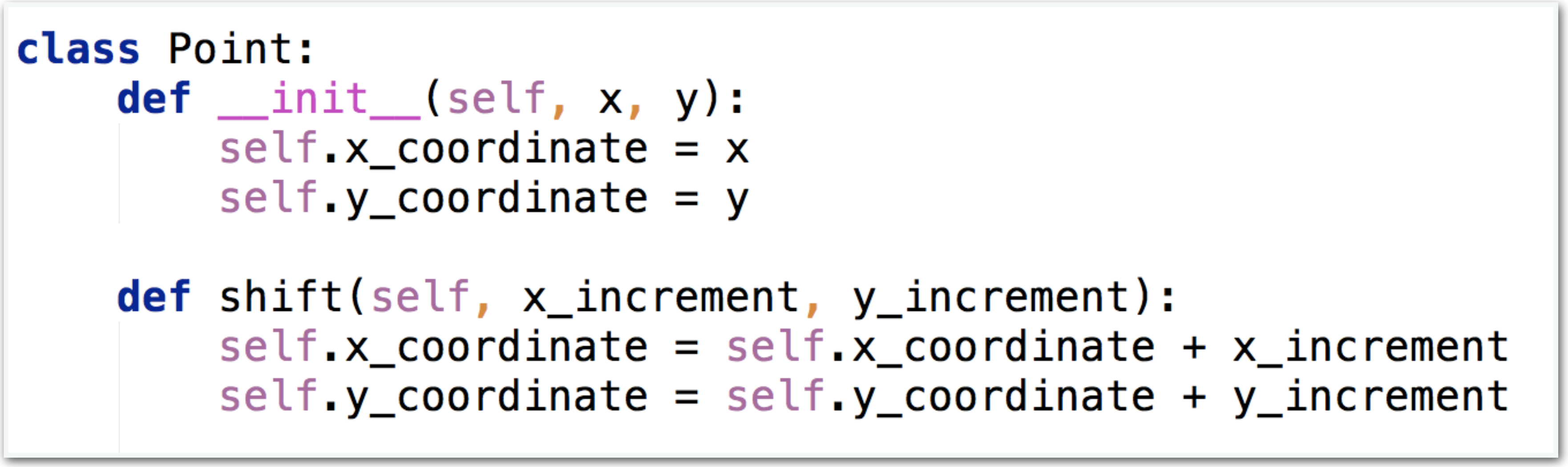




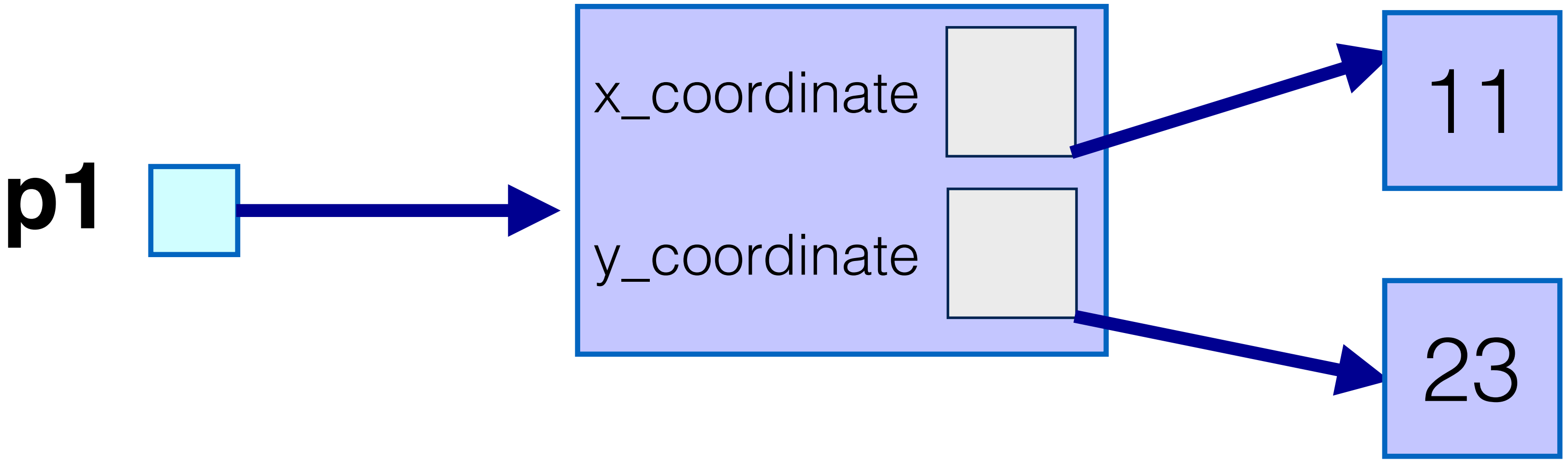
```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```

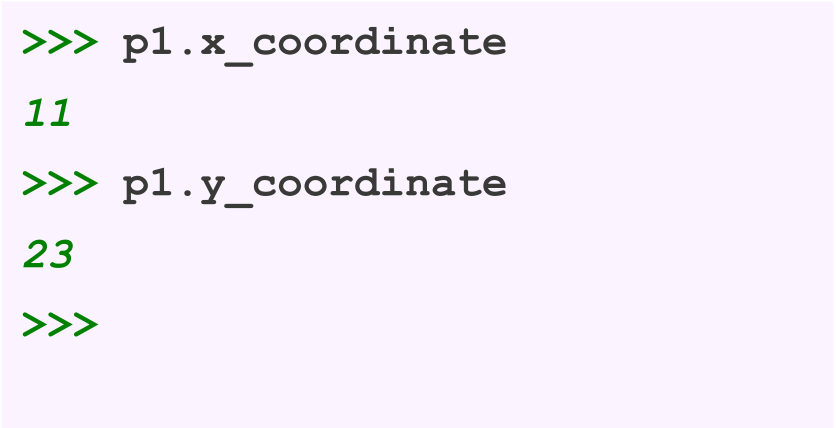


Once the function finishes executing the function **namespace** is gone.



```
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y

    def shift(self, x_increment, y_increment):
        self.x_coordinate = self.x_coordinate + x_increment
        self.y_coordinate = self.y_coordinate + y_increment
```





```
>>> p1.x_coordinate
11
>>> p1.y_coordinate
23
>>>
```

Once the function finishes executing the function **namespace** is gone.

Binding a name

- There are many ways to **bind a name** in Python
- For example, by:
  - **Assigning** to a variable (x = 13)
  - Receiving an **argument** (e.g., for *x_increment* and *y_increment*)
  - Importing a **module** (**import x**)
  - Importing a variable (from y import x)
  - Defining a **function** (**def x**(foo): ...)
  - **Defining a class** (class x: ...)
  - Writing a for loop (for x in y: ...)
  - Writing an except clause (try: ... except x: ...)
- If any of these appears inside a function. It makes the name local to the function

```
>>> x = 'first'
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
```

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
```

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
```

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
```

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
```

a() has **x** in its local namespace with value **a**

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
```

b() does not have **x** in its local namespace. It looks at next level where it finds x with value **‘first’**

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
```

as before, but now the value of global **x** is **‘’second’**

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
```

as before: **x** is in the local namespace of a()
with value **‘a’**

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
>>> c(7)
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
>>> c(7)
7
```

c() has **x** in its local namespace with value **7**

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
>>> c(7)
7
>>> d()
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
>>> c(7)
7
>>> d()
second
```

d() calls **b()**, which is as before

```
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
>>> c(7)
7
>>> d()
second
>>> e()
>>> x = 'first'
>>> def a():
...     x = 'a'
...     print(x)
...
>>> def b():
...     print(x)
...
>>> def c(x):
...     print(x)
...
>>> def d():
...     x = 'd'
...     b()
...
>>> def e():
...     x = 'e'
...     def f():
...         print(x)
...     f()
```

```
>>> a()
a
>>> b()
first
>>> x = 'second'
>>> b()
second
>>> a()
a
>>> c(7)
7
>>> d()
second
>>> e()
e
```

e() defines and calls **f()**, which does not have **x** in its local namespace. so it looks in the namespace of the enclosing function and finds it with value **‘e’**.

Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to “qualify” the name.

Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to “qualify” the name.

```
>>> import point
>>> p1 = point.Point(1,3)
```

Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to “qualify” the name.

```
>>> import point
>>> p1 = point.Point(1,3)
```

qualify a name

Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to "qualify" the name.

```
>>> import point
>>> p1 = point.Point(1,3)
```

qualify a name

```
>>> from point import Point
>>> p1 = Point(1,3)
```

Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to “qualify” the name.

```
>>> import point
>>> p1 = point.Point(1,3)
```

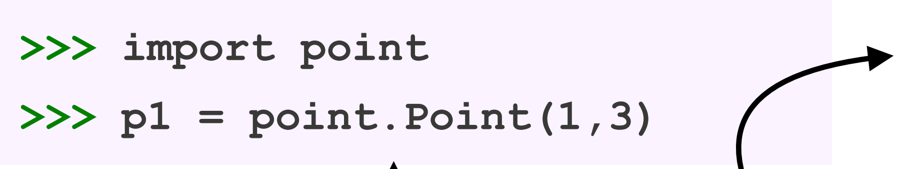
qualify a name

```
>>> from point import Point
>>> p1 = Point(1,3)
```

Brings Point to the **current namespace**
Then, there is **no need** for *point.Point*

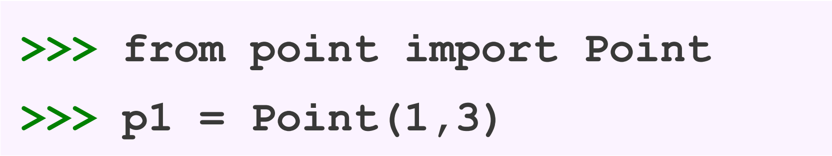
Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to "qualify" the name.



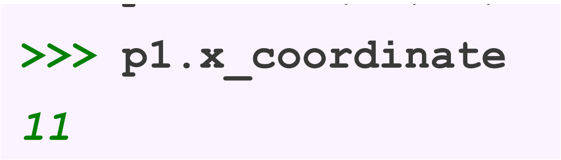
```
>>> import point
>>> p1 = point.Point(1,3)
```

qualify a name



```
>>> from point import Point
>>> p1 = Point(1,3)
```

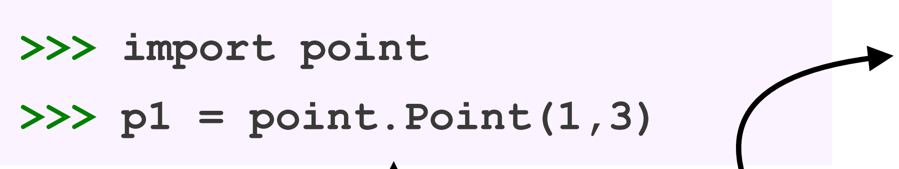
Brings Point to the **current namespace**
Then, there is **no need** for *point.Point*



```
>>> p1.x_coordinate
11
```

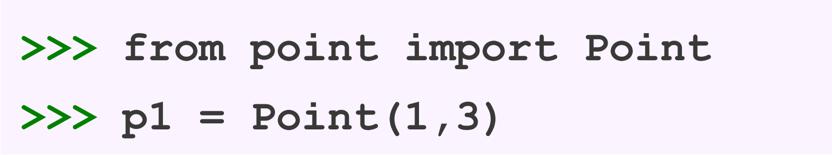
Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to "qualify" the name.



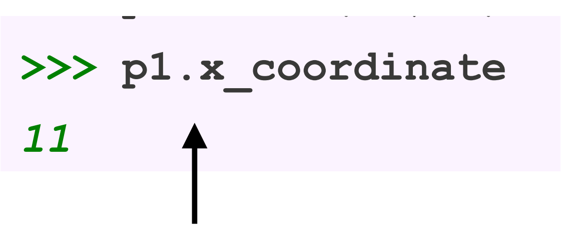
```
>>> import point
>>> p1 = point.Point(1,3)
```

qualify a name



```
>>> from point import Point
>>> p1 = Point(1,3)
```

Brings Point to the **current namespace**
Then, there is **no need** for *point.Point*

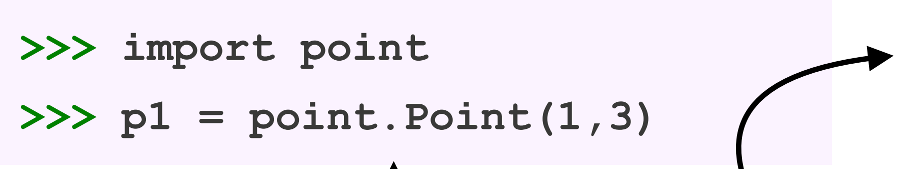


```
>>> p1.x_coordinate
11
```

qualify a name

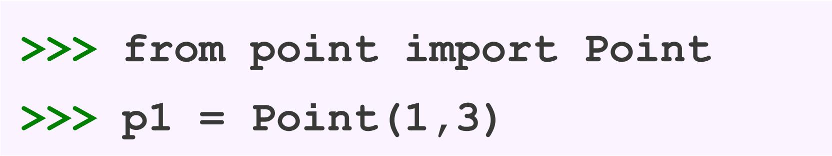
Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to “qualify” the name.



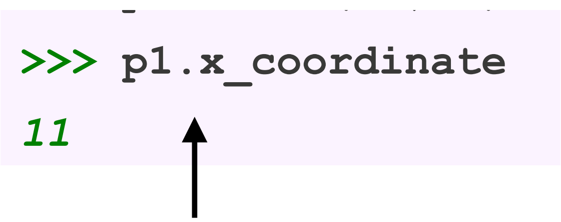
```
>>> import point
>>> p1 = point.Point(1,3)
```

qualify a name



```
>>> from point import Point
>>> p1 = Point(1,3)
```

Brings Point to the **current namespace**
Then, there is **no need** for *point.Point*



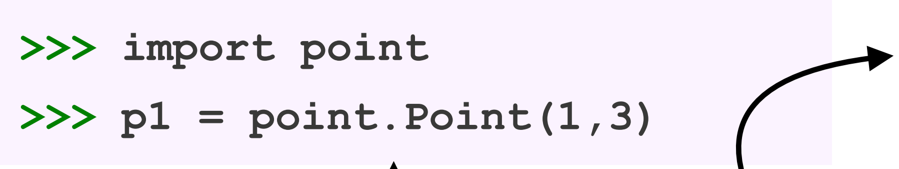
```
>>> p1.x_coordinate
11
```

qualify a name

- Often there are **several scopes in operation**:
  - The scope of the **method** that is executing
  - The scope of the **class** where the method is defined
  - The scope of the **module** where the class is defined
  - The scope of the **interpreter** that is executing

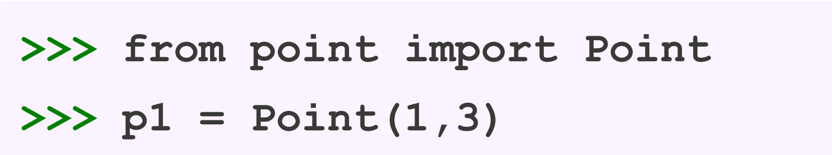
Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to "qualify" the name.



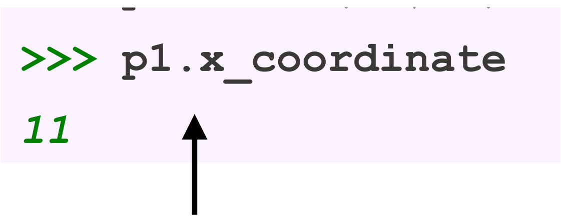
```
>>> import point
>>> p1 = point.Point(1,3)
```

qualify a name



```
>>> from point import Point
>>> p1 = Point(1,3)
```

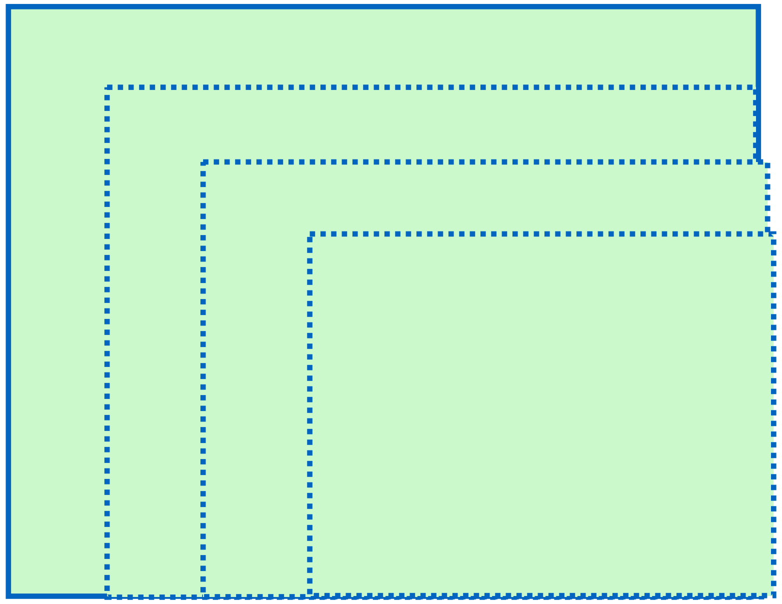
Brings Point to the **current namespace**
Then, there is **no need** for *point.Point*



```
>>> p1.x_coordinate
11
```

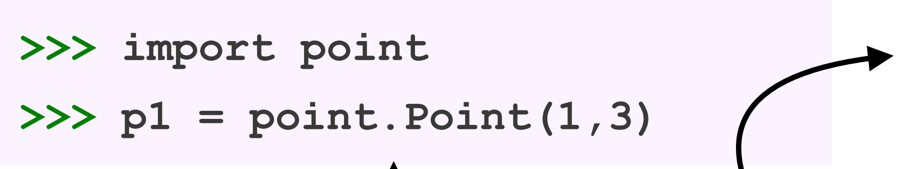
qualify a name

- Often there are **several scopes in operation**:
  - The scope of the **method** that is executing
  - The scope of the **class** where the method is defined
  - The scope of the **module** where the class is defined
  - The scope of the **interpreter** that is executing



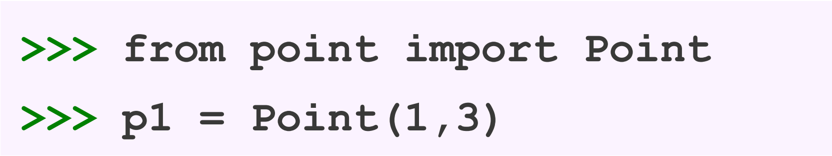
Scoping

- **Scope**: block of text where a namespace is directly accessible. That is, where there is no need to “qualify” the name.



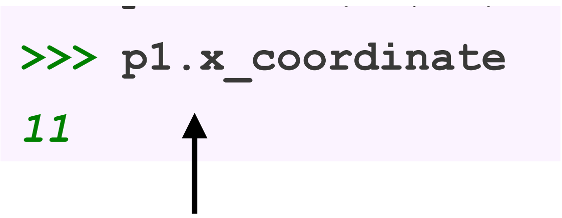
```
>>> import point
>>> p1 = point.Point(1,3)
```

qualify a name



```
>>> from point import Point
>>> p1 = Point(1,3)
```

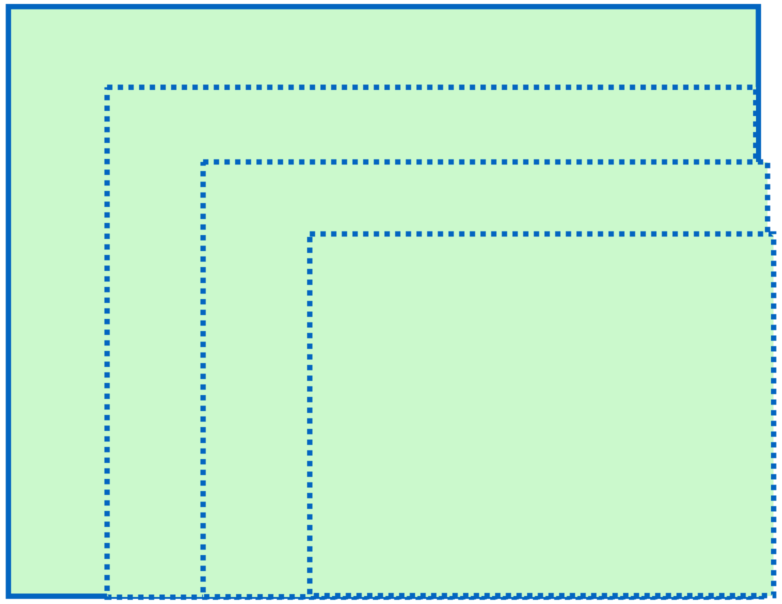
Brings Point to the **current namespace**
Then, there is **no need** for *point.Point*



```
>>> p1.x_coordinate
11
```

qualify a name

- Often there are **several scopes in operation**:
  - The scope of the **method** that is executing
  - The scope of the **class** where the method is defined
  - The scope of the **module** where the class is defined
  - The scope of the **interpreter** that is executing



- Scope is **determined statically** but **used dynamically**
  - Statically: that you can always determine the scope of any name by looking at the program
  - Dynamically: that it is at run-time that Python searches for names

Scoping Rules

- **Names belong to the namespace** where they are bound. The scope of a name does not change while the program is running.

Scoping Rules

- **Names belong to the namespace** where they are bound. The scope of a name does not change while the program is running.

- During execution, Python searches for names as follows:

Scoping Rules

- **Names belong to the namespace** where they are bound. The scope of a name does not change while the program is running.

- During execution, Python searches for names as follows:
  - First, in the **innermost** scope
    - Contains all the local names (those in the method’s namespace)

Scoping Rules

- **Names belong to the namespace** where they are bound. The scope of a name does not change while the program is running.

- During execution, Python searches for names as follows:
  - First, in the **innermost** scope
    - Contains all the local names (those in the method's namespace)
  - Then, in the scopes of any **enclosing** functions:
    - Searched from the nearest-to-outer enclosing scope
    - Contains nonlocal and nonglobal names

Scoping Rules

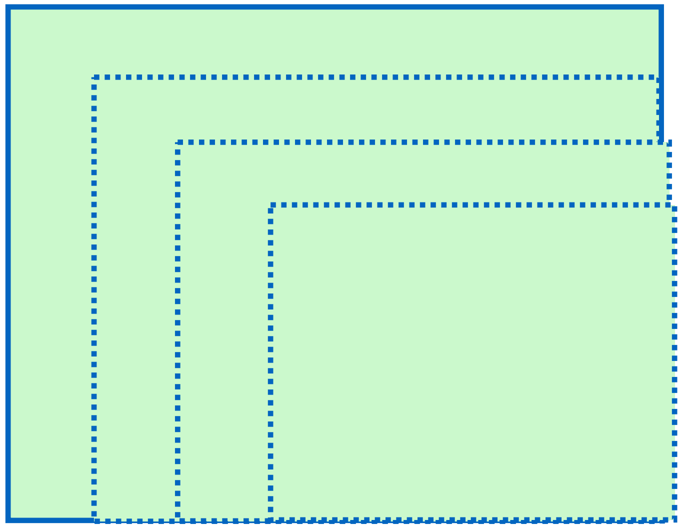
- **Names belong to the namespace** where they are bound. The scope of a name does not change while the program is running.

- During execution, Python searches for names as follows:
  - First, in the **innermost** scope
    - Contains all the local names (those in the method's namespace)
  - Then, in the scopes of any **enclosing** functions:
    - Searched from the nearest-to-outer enclosing scope
    - Contains nonlocal and nonglobal names
  - Then the current **module's global names**
    - That is, those in the module's namespace

Scoping Rules

- **Names belong to the namespace** where they are bound. The scope of a name does not change while the program is running.

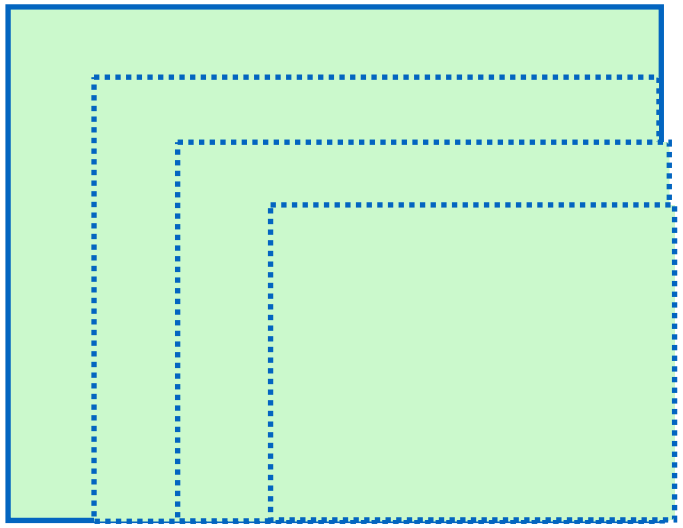
- During execution, Python searches for names as follows:
  - First, in the **innermost** scope
    - Contains all the local names (those in the method's namespace)
  - Then, in the scopes of any **enclosing** functions:
    - Searched from the nearest-to-outer enclosing scope
    - Contains nonlocal and nonglobal names
  - Then the current **module's global names**
    - That is, those in the module's namespace
  - Last, the namespace containing **built-in names**



Scoping Rules

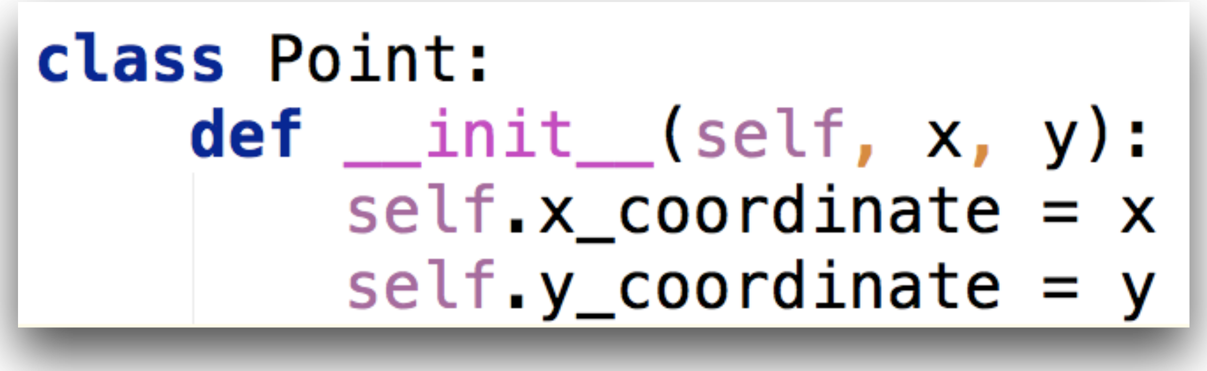
- **Names belong to the namespace** where they are bound. The scope of a name does not change while the program is running.

- During execution, Python searches for names as follows:
  - First, in the **innermost** scope
    - Contains all the local names (those in the method's namespace)
  - Then, in the scopes of any **enclosing** functions:
    - Searched from the nearest-to-outer enclosing scope
    - Contains nonlocal and nonglobal names
  - Then the current **module's global names**
    - That is, those in the module's namespace
  - Last, the namespace containing **built-in names**

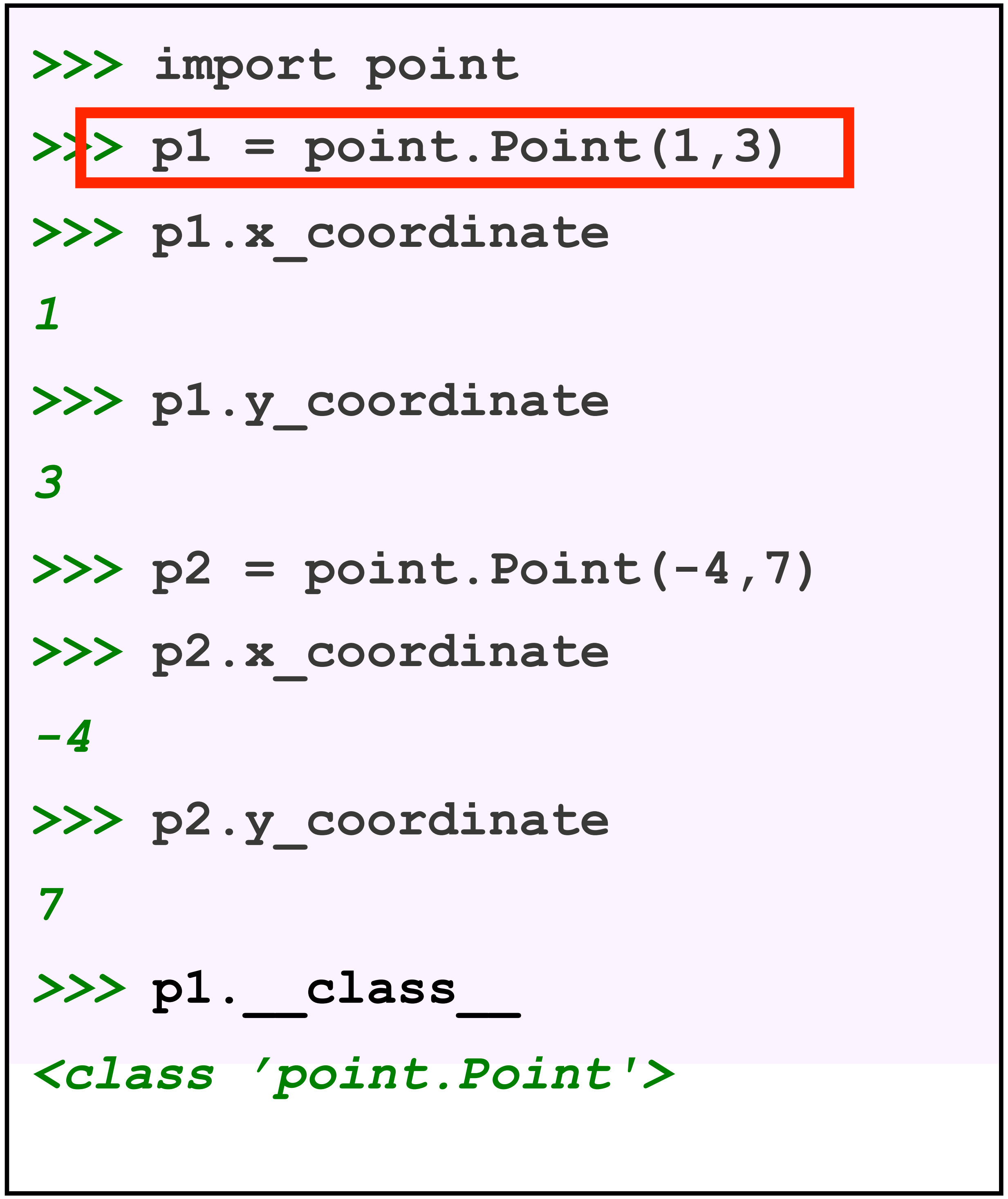


- Programmers can change the scope of identifiers. But we are not going to see this.

“Qualifying”

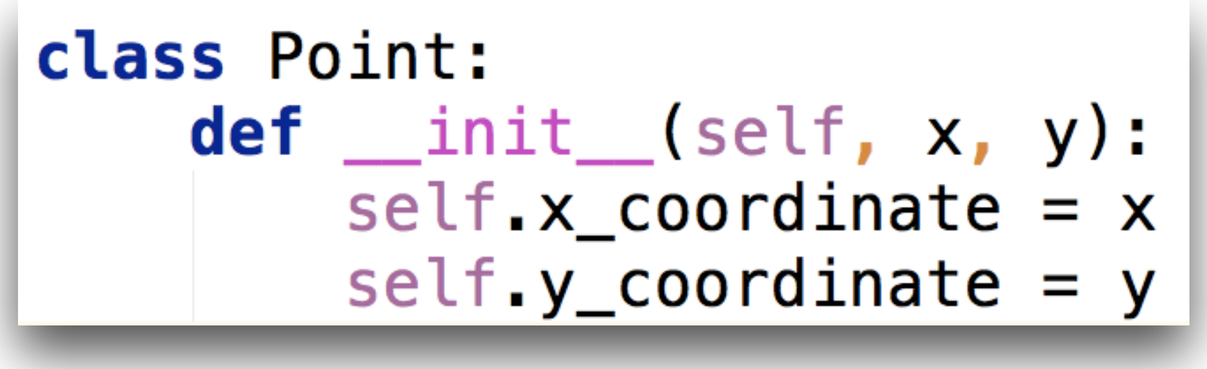


```
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y
```



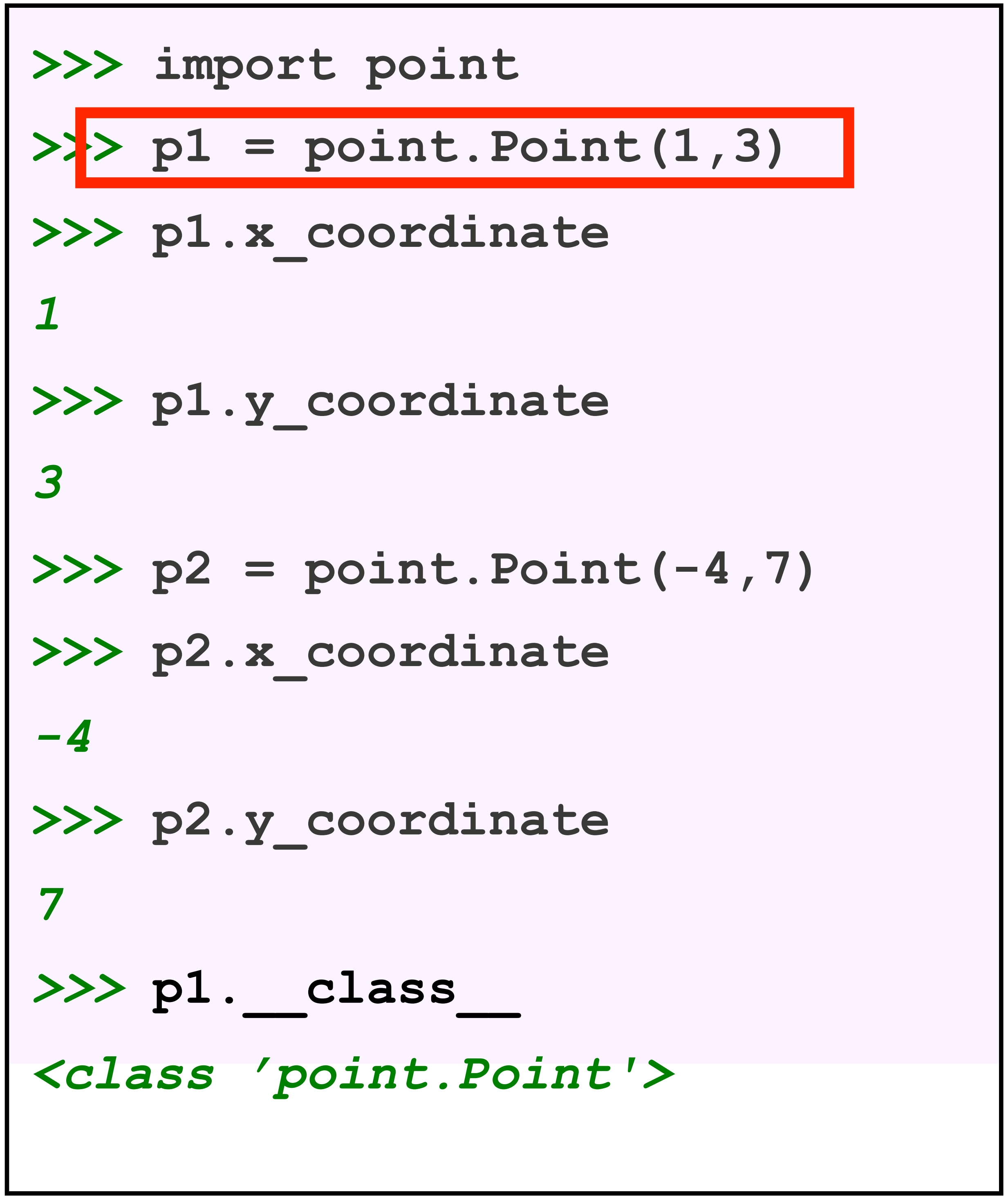
```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.x_coordinate
1
>>> p1.y_coordinate
3
>>> p2 = point.Point(-4,7)
>>> p2.x_coordinate
-4
>>> p2.y_coordinate
7
>>> p1.__class__
<class 'point.Point'>
```

“Qualifying”



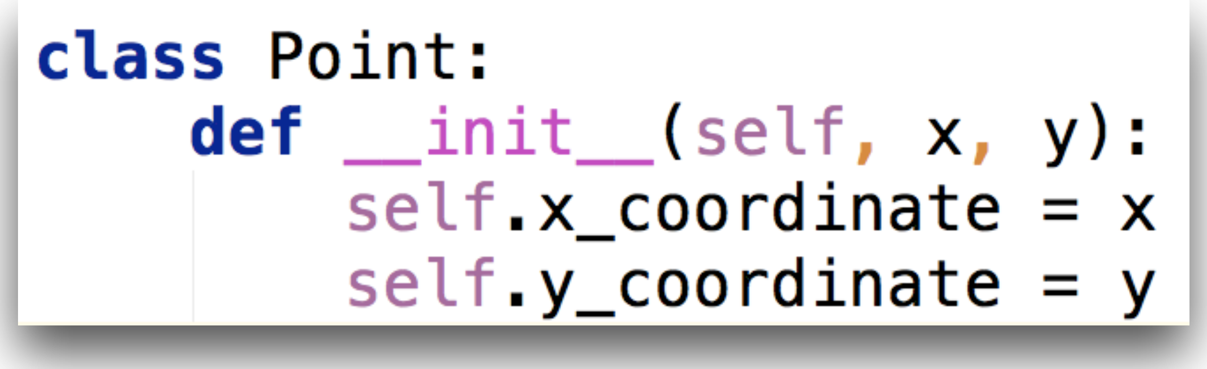
```
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y
```

Why is **.point** needed?



```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.x_coordinate
1
>>> p1.y_coordinate
3
>>> p2 = point.Point(-4,7)
>>> p2.x_coordinate
-4
>>> p2.y_coordinate
7
>>> p1.__class__
<class 'point.Point'>
```

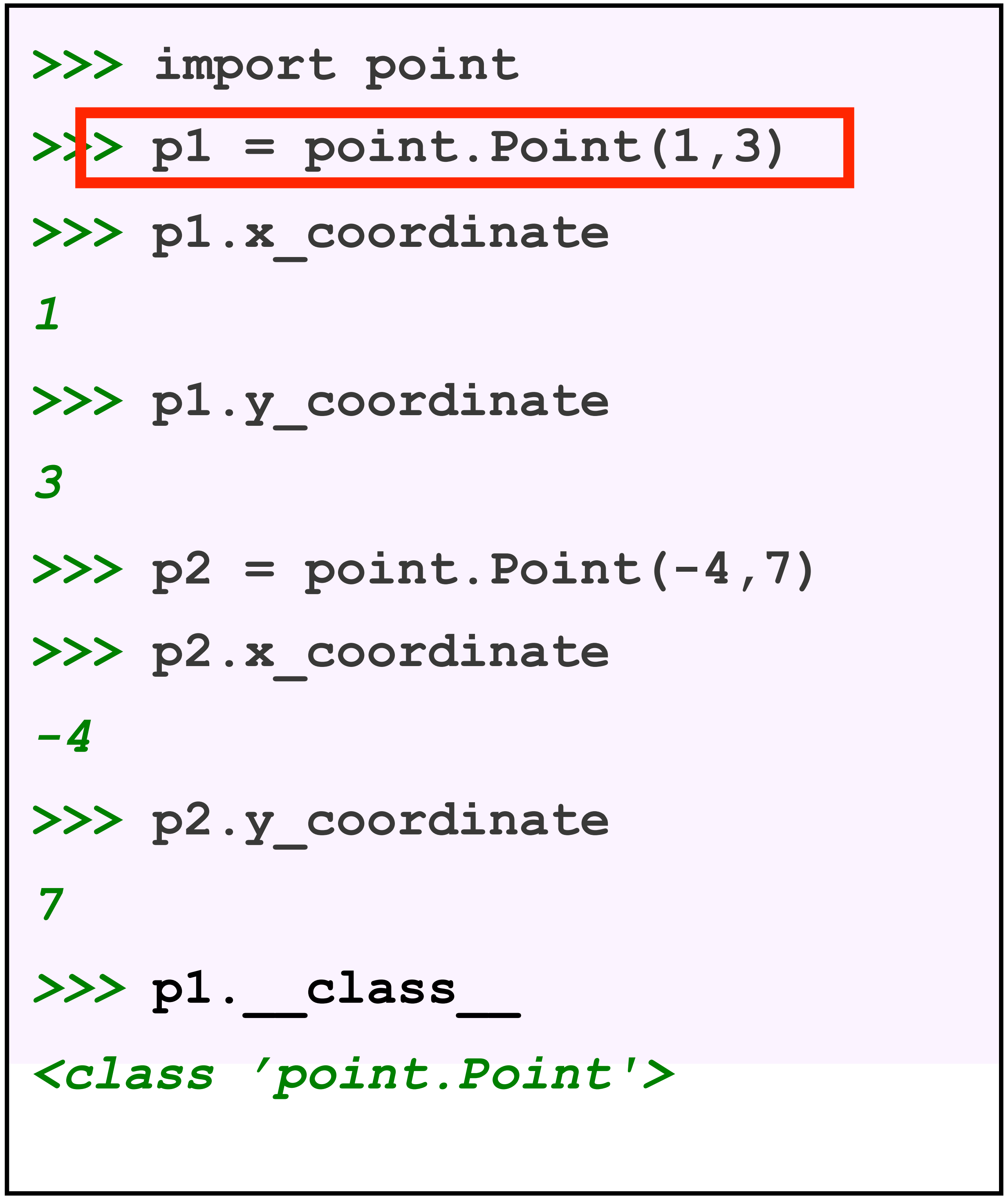
“Qualifying”



```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y
```

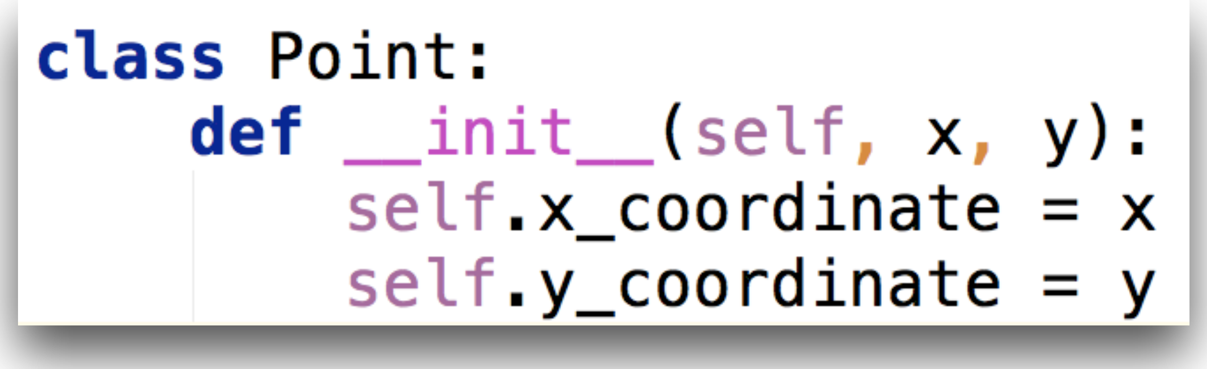
Why is **.point** needed?

The name **`Point`** is not directly accessible from the current code, i.e., not in its namespace or in any one where Python will search for it



```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.x_coordinate
1
>>> p1.y_coordinate
3
>>> p2 = point.Point(-4,7)
>>> p2.x_coordinate
-4
>>> p2.y_coordinate
7
>>> p1.__class__
<class 'point.Point'>
```

“Qualifying”

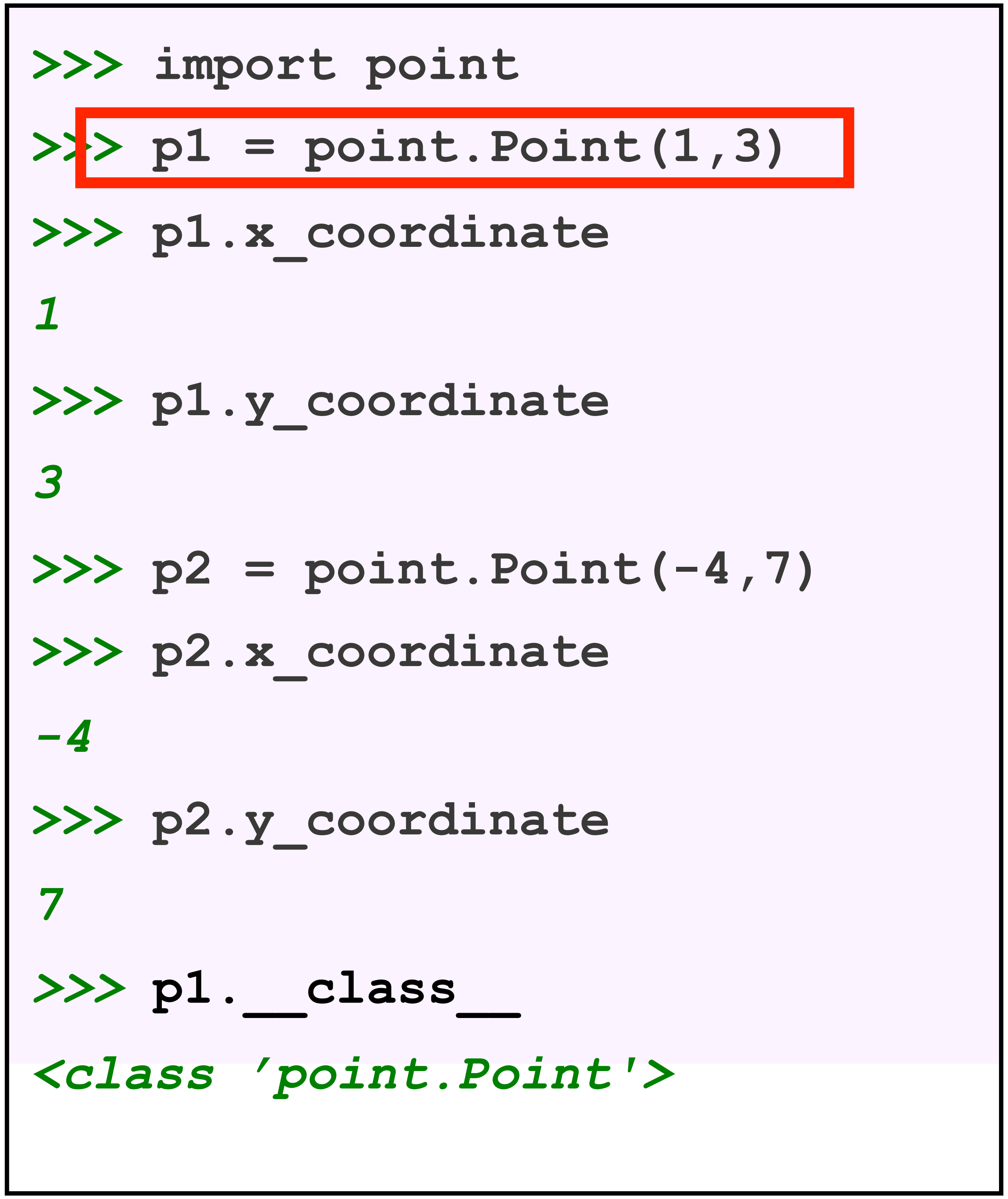


```python
class Point:
    def __init__(self, x, y):
        self.x_coordinate = x
        self.y_coordinate = y
```

Why is **.point** needed?

The name **`Point`** is not directly accessible from the current code, i.e., not in its namespace or in any one where Python will search for it

Qualifying it by **`point.`** allows us to access the namespace of module **`point.`** which contains the name **`Point`**



```
>>> import point
>>> p1 = point.Point(1,3)
>>> p1.x_coordinate
1
>>> p1.y_coordinate
3
>>> p2 = point.Point(-4,7)
>>> p2.x_coordinate
-4
>>> p2.y_coordinate
7
>>> p1.__class__
<class 'point.Point'>
```

Remember Silly

```
>>> class Silly:
...     i = 8
...
>>> Silly.i
8
>>> s1 = Silly()
>>> s1.i
8
>>> s2 = Silly()
>>> s2.i
8
>>> Silly.i = 11
```

```
>>> s1.i
11
>>> s2.i
11
>>> s1.i = 6
>>> s1.i
6
>>> s2.i
11
>>> Silly.i = 22
>>> s1.i
6
>>> s2.i
22
```

Remember Silly

```
>>> class Silly:
...       i = 8
...
>>> Silly.i
8
>>> s1 = Silly()
>>> s1.i
8
>>> s2 = Silly()
>>> s2.i
8
>>> Silly.i = 11
```

```
>>> s1.i
11
>>> s2.i
11
>>> s1.i = 6
>>> s1.i
6
>>> s2.i
11
>>> Silly.i = 22
>>> s1.i
6
>>> s2.i
22
```

?
shouldn’t this be **6** since **i** is a **class variable**?

Remember Silly

```
>>> class Silly:
...       i = 8
...
>>> Silly.i
8
>>> s1 = Silly()
>>> s1.i
8
>>> s2 = Silly()
>>> s2.i
8
>>> Silly.i = 11
```

```
>>> s1.i
11
>>> s2.i
11
>>> s1.i = 6
>>> s1.i
6
>>> s2.i
11
>>> Silly.i = 22
>>> s1.i
6
>>> s2.i
22
```

Creates a new attribute for **s1** (but **not for s2**) in its local namespace

?
shouldn't this be **6** since **i** is a **class variable**?

Remember Silly

```
>>> class Silly:
...     i = 8
...
>>> Silly.i
8
>>> s1 = Silly()
>>> s1.i
8
>>> s2 = Silly()
>>> s2.i
8
>>> Silly.i = 11
```

```
>>> s1.i
11
>>> s2.i
11
>>> s1.i = 6
>>> s1.i
6
>>> s2.i
11
>>> Silly.i = 22
>>> s1.i
6
>>> s2.i
22
```

Creates a new attribute for **s1** (but **not for s2**) in its local namespace

?
shouldn't this be **6** since **i** is a **class variable**?

First looks in the **local namespace**.

Summary

- We have seen how to **draw memory diagrams** for code involving:
    - Variable **assignments**
    - **Mutable** types
    - **Immutable** types
    - Assigning variables to other variables ("**variable aliasing**")