Introduction to Box-and-Pointer Diagrams

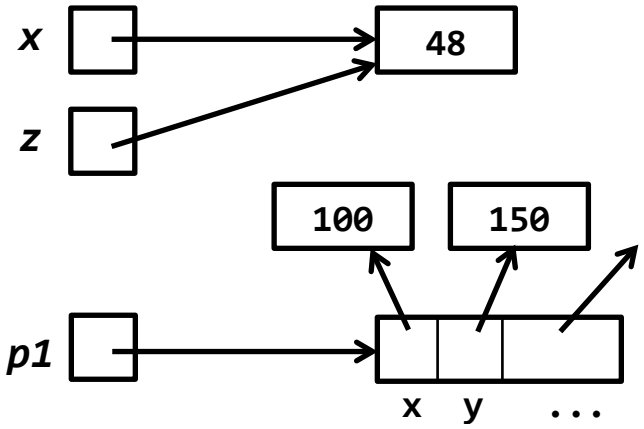
A ***Box-and-Pointer Diagram*** displays a ***snapshot*** of the ***variables and their values*** at a given point of execution of a program.

For example, after the code below executes:

```
x = 48
z = x
p1 = zg.Point(100, 150)
```

the corresponding box-and-pointer diagram is as shown to the right. (We will work through the details of this and other examples in this exercise.)

Box and Pointer diagram:



In this reading and the exercise that accompanies it, you will learn ***how to draw and read box-and-pointer diagrams***. Additionally, you will see how they clarify the following key ideas (all of which you'll examine in this exercise):

- ***Variables are REFERENCES to objects.***
- ***Assignment*** (e.g. `x = 100`) causes a variable to refer to an object.
- ***Function calls*** (e.g. `foo(54, x)`) also cause variables to refer to objects.
- Some types of objects are ***mutable***, which means that the "insides" of such an object can be re-assigned. (We'll be more explicit shortly.) Other types of objects are ***immutable***.

Objects, Variables and References

Recall that in Python, every piece of data is an ***object***. Objects have a ***type*** (which determines what the object can do and the types of data that the object contains/knows) and a ***value***.

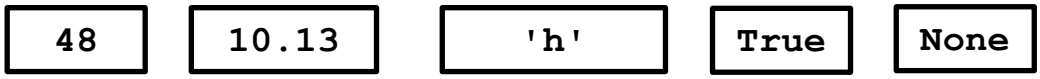
Variables are REFERENCES to objects – we say that the variable ***refers*** to an object, or ***points*** to an object, or is the ***name*** for an object (several ways to say the same thing, in this context).

Some objects contain ***references to other objects***; these objects are called ***containers***. A **zg.Point** is a container (as is any object of a user-defined class). **Lists**, **strings** and **tuples** are containers. **Numbers** are **not** containers.

We will see that box-and-pointer diagrams help clarify what we mean by ***references***.

How to Draw a Box-and-Pointer Diagram

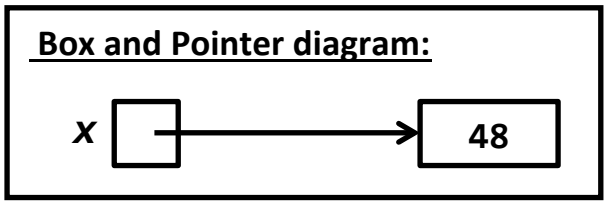
Rule 1: ***Draw a NON-CONTAINER OBJECT by putting its value inside a box***, as in these examples:



Rule 2: ***Draw a VARIABLE using a box labeled with the variable's name*** and with ***arrows from the box to the object*** to which the variable currently refers.

For example, after this code executes:

```
x = 48
```



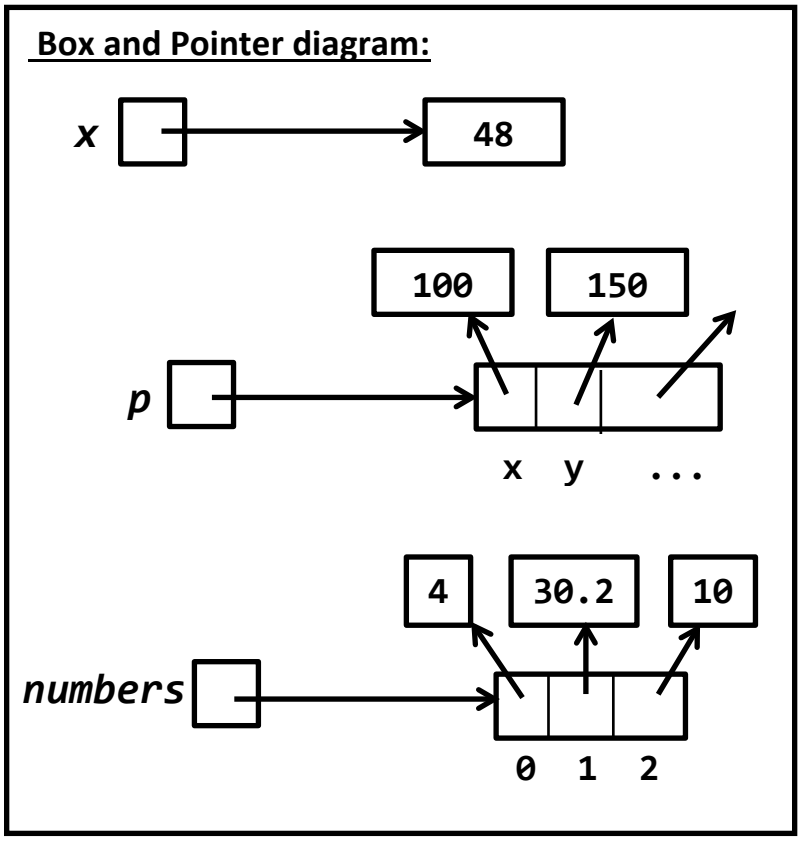
the corresponding box-and-pointer diagram is as shown above and to the right.

Rule 3: ***Draw a CONTAINER OBJECT by making a box for it, and then creating sub-boxes*** that are drawn ***as if they were variables***, but with ***field names*** for fields of an object and ***indices*** for items of a sequence.

For example, after this code executes:

```
x = 48
p = zg.Point(100, 150)
numbers = [4, 30.2, 10]
```

the corresponding box-and-pointer diagram is as shown to the right. The ellipsis (…) for the zg.Point object reflects the fact that zg.Points have an outline color, etc.

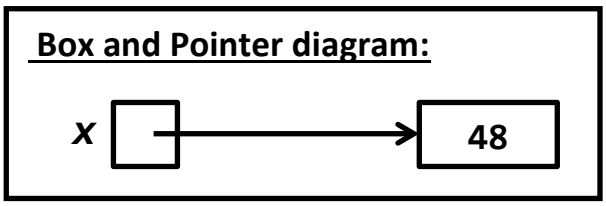


Re-assigning a variable

Recall that ***variables are REFERENCES to objects*** – we say that the variable ***refers*** to an object, or ***points*** to an object, or is the ***name*** for an object (several ways to say the same thing, in this context). Box-and-pointer diagrams help clarify what we mean by ***references***, as in the following example:

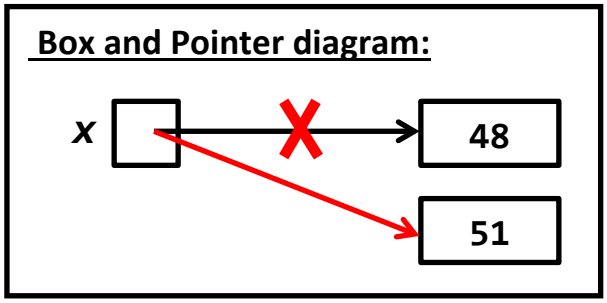
The box-and-pointer diagram on the right shows the situation after the code on the left runs.

```
x = 48
```



If we now ***RE***-assign the variable **x**, as shown in the example to the right, the arrow out of the box for ***x*** changes – it points to the new value for ***x***.

```
x = x + 3
```



Note that we put an ✘ through the old arrow to indicate that the old arrow no longer exists.

We choose ***not*** to ***erase*** arrows (instead, we ✘ them out) so that the diagram can show the ***history*** of the changes over time as the lines of code execute.

(This continues on the next page.)

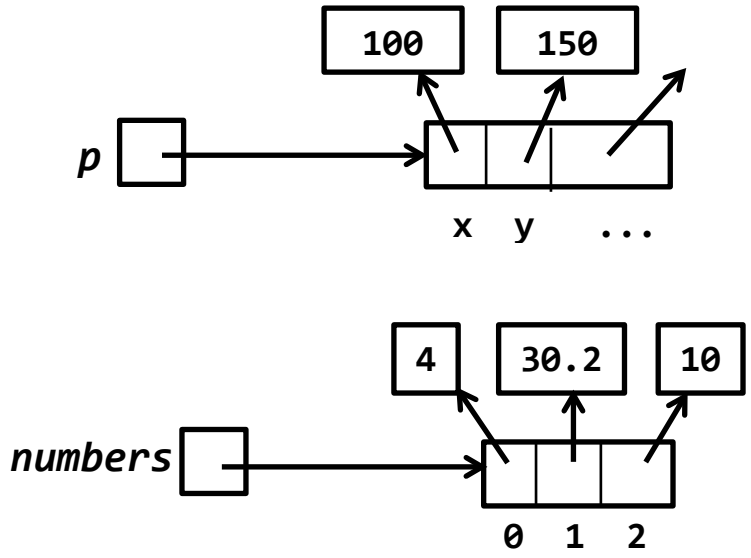
Mutable objects

Some objects contain ***references to other objects***; these objects are called ***containers***. A **zg.Point** is a container (as is any object of a user-defined class). **Lists**, **strings** and **tuples** are containers. **Numbers** are **not** containers.

Here is an example showing assignments of two container objects, with the corresponding box-and-pointer diagram shown to the right:

```
p = zg.Point(100, 150)
numbers = [4, 30.2, 10]
```

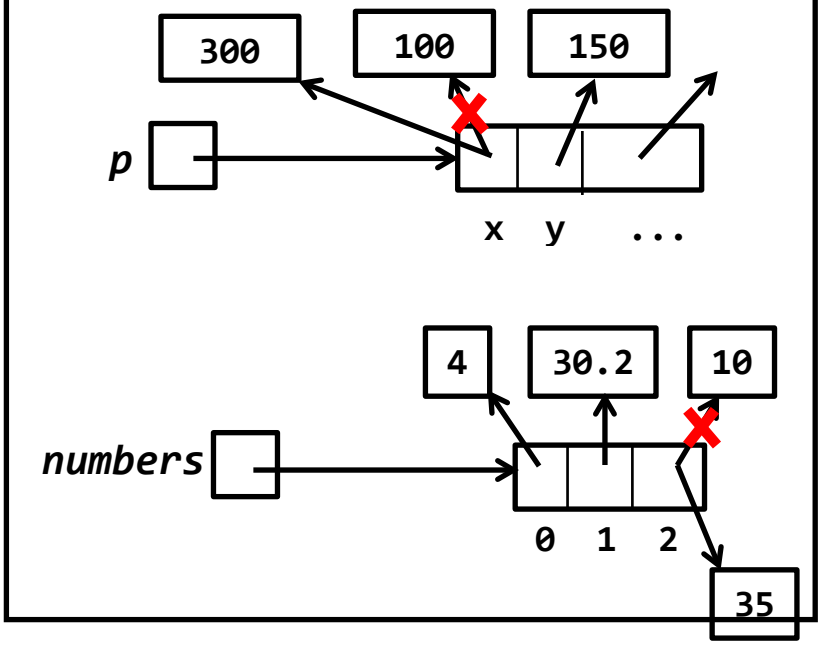
Box and Pointer diagram:



Suppose that we do ***RE***-assignments of the insides of the container objects, like this:

```
p.x = 300
numbers[2] = number[0] + 31
```

Box and Pointer diagram:



The corresponding box-and-pointer diagram is shown to the right. Note that the arrows from ***p*** and ***numbers*** did NOT change; rather, the arrows from items "inside" the ***zg.Point*** and ***list*** objects change.

This sort of assignment has a name – we say that ***p*** and ***numbers*** were ***mutated***. That means that although they still point to the same objects to which they previously pointed, the ***insides*** of those (container) objects have changed (i.e., point to new values).

Mutable objects allow efficient use of time and space but are dangerous

Mutable objects are good because they allow for more efficient use of space and time:

Suppose that you have a container object that has thousands (or more) of items inside it.

- If the container object is mutable, you can change a single item inside the container without changing the rest of the items inside the container.
- The alternative would be to copy the entire object, with its thousands (or more) items, modifying the single item within as you do so, and use the copy (keeping the original intact).

The alternative-to-mutation approach requires twice as much space as the mutation approach, since you have to copy all the items in the container object. Additionally, there is a cost in time since you must take the time to do all those copies.

Mutable objects are dangerous:

Suppose you send an immutable object (e.g. a number) to a function. When the function returns, you are guaranteed that that object has not changed – good! But if you send a ***mutable*** object to a function, there is no such guarantee – bad! For example, suppose that the function (which someone else wrote) has a bug and it accidentally destroys part of the object. Then your code might be completely correct, yet the program as a whole might have a serious security flaw simply because of the faulty function.

Which objects are mutable and which objects are immutable?

1. Only container objects are (potentially) mutable. So numbers, which are NOT containers, are immutable.
2. Python provides two general-purpose ***sequence*** types: ***lists*** and ***tuples***. Both can contain objects of any type. Their only difference (other than notation) is that:
   - ***Lists*** are ***mutable***.
   - ***Tuples*** are ***immutable***.

   So, if you need the possibility of efficient use of space and time when the sequence may change, use a list. But if you know the sequence will not change (maybe it is used in a "read-only" form), or if you need the security of immutability (maybe it is a life-critical application for which efficiency is not a factor), use a tuple.
3. ***Strings*** are ***immutable***.
4. ***Any object of a user-defined class*** (e.g. ***`zg.Point`***) is ***mutable***.[1]

[1] Exception: Classes that extend immutable-type classes like ***`tuple`*** will themselves be immutable-type classes. We will not see such classes in this course.