**6.2 Assignment Statements and Copying**

Assignment statements basically assign a value to a variable. Another way of saying this is that the assignment binds the variable to the value.

We have seen that variables refer to locations in which values are stored. What happens when one variable is assigned to another variable, e.g.

*var2 = var1*

is that they both refer to the same location. This has interesting ramifications when modifying variables, especially for mutable types.

Assigning an expression (not another variable) to a variable in assignment statements creates a new location for the variable to reference, and stores the result of the expression in that location. Assigning another variable, however, does not create a new location; instead, both variables now refer to the same location.

Consider the following code:

*>>> x = 4*

*>>> y = x*

What happens is that the variable *x* is created, and it refers to a location in which a 4 is stored. Then, assigning *x* to *y* means that *y* is now also going to refer to the same location as *x*. We can picture this as follows:

|  |
| --- |
| y |
| x |
| 4 |

Then, assume that the following assignment statement is executed:

*>>> x = 5*

This creates a new location in which a 5 is stored, and *x* now refers to that new location. The variable *y*, however, has not been changed. It still refers to the location in which 4 is stored.

*>>> x*

5

*>>> y*

4

This may seem like a nuance, but it is very important especially for mutable types.

In the next section, operators and functions that illustrate how this works will be introduced.

**6.2.1 Variable Identities**

Before we examine assigning one variable to another, let us first revisit assignment statements and how they work. One important piece of information is that every location that a variable references is assigned a unique **identity**. The value of this identity can be found with the **id** function. In some cases the identity is the actual address of the location that the variable refers to. Regardless of whether this is true or not, every variable has a unique identity number. Do not try to make sense of the actual numbers!

*>>> varname = 11*

*>>> print(id(varname))*

140662851902064

What actually happens when we assign a value to a variable, and then reassign a different value to that variable? We can print the value of the variable and its identity to see what happens.

*>>> var1 = 5.3*

*>>> print(var1, id(var1))*

5.3 140663007004368

*>>> var1 = 11.11*

*>>> print(var1, id(var1))*

11.11 140663007003760

We can see that the value changed, but the identity did, also. This means that the second assignment did not actually modify what was stored in the original location referred to by *var1*, but rather created a new location.

This happens also if we modify the value of the variable by multiplying the current value by 2.

*>>> var1 = 5.3*

*>>> print(var1, id(var1))*

5.3 140663007003280

*>>> var1 = var1 \* 2*

*>>> print(var1, id(var1))*

10.6 140663007003888

Again, the second assignment created a new location and stored the result of the expression on the right (var1 \* 2) in that location, and *var1* now refers to the new location.

Let’s now see what happens when we assign one number variable to another. We can see the locations using the **id** function. However, when comparing two identities, it is simpler to use the **is** operator. The **is** operator can be used to determine whether two variables refer to the same location, or in other words, whether their identities are the same or not. The expression *a is b* results in **True** if a and b refer to the same location (have the same identities) or **False** if not.

*radius = 3*

*rad = radius*

*print(radius, id(radius))*

*print(rad, id(rad))*

*if rad is radius:*

*print('They are the same')*

*else:*

*print('They are not the same')*

*print()*

*radius = 5*

*print(radius, id(radius))*

*print(rad, id(rad))*

*if rad is radius:*

*print('They are the same')*

*else:*

*print('They are not the same')*

3 140437299009904

3 140437299009904

They are the same

5 140437299009968

3 140437299009904

They are not the same

We can see that after the initial assignment *rad = radius* the two variable names referred to the same location. However, the assignment *radius = 5* then created a new location, a new identifier, for the variable *radius* to refer to.

There is also an **is not** operator. The expression

a is not b

is equivalent to

not (a is b)

but is easier to read and understand.

The same behavior can be seen for other simple types like Boolean, and for immutable types such as strings and tuples, as shown in the following simplified examples:

*>>> relate = 6 < 10*

*>>> isit = relate*

*>>> print(relate, isit)*

*>>> isit = 6 > 10*

*>>> print(relate, isit)*

True True

True False

*>>> word = "hello"*

*>>> greetword = word*

*>>> print(word, greetword)*

*>>> word = "hi"*

*>>> print(word, greetword)*

hello hello

hi hello

*>>> tupone = (5, 11, 'x')*

*>>> tuptwo = tupone*

*>>> print(tupone, tuptwo)*

*>>> tupone = (4.4, 39)*

*>>> print(tupone, tuptwo)*

(5, 11, 'x') (5, 11, 'x')

(4.4, 39) (5, 11, 'x')

For lists, assigning a new list to one of the list variables results in the same behavior that has been seen with other types.

*listone = [5, 11, 'x']*

*listtwo = listone*

*print(listone, listtwo)*

*if listtwo is listone:*

*print('They are the same')*

*else:*

*print('They are not the same')*

*print()*

*listone = ['q', '?', 0]*

*print(listone, listtwo)*

*if listtwo is listone:*

*print('They are the same')*

*else:*

*print('They are not the same')*

[5, 11, 'x'] [5, 11, 'x']

They are the same

['q', '?', 0] [5, 11, 'x']

They are not the same

This is because again, all assignments create a new location for the variable to refer to. However, lists are different in that they are mutable. This means that the contents of a list can be modified. Changing the contents of a list variable is different from assigning a brand new list!

*listone = [5, 11, 'x']*

*listtwo = listone*

*print(listone, listtwo)*

*if listtwo is listone:*

*print('They are the same')*

*else:*

*print('They are not the same')*

*print()*

*listone[1] = 27*

*print(listone, listtwo)*

*if listtwo is listone:*

*print('They are the same')*

*else:*

*print('They are not the same')*

[5, 11, 'x'] [5, 11, 'x']

They are the same

[5, 27, 'x'] [5, 27, 'x']

They are the same

This means that the two list variables refer to the same location, and the contents of one of the items in the list was modified – but no new lists were created, so the variables still referred to the same location at the end of the code. This same behavior will occur any time a list is modified.

We have seen that in Python, for the types that we have covered so far, assigning one variable to another creates an alternate reference to the same location.

Beware of list slices, however! Slicing a list using just the colon is different from using the variable name.

*>>> listone = [5, 11, 'x']*

*>>> listtwo = listone*

*>>> listthree = listone[:]*

*>>> print(listone, listtwo, listthree)*

*>>> print(id(listone), id(listtwo), id(listthree))*

[5, 11, 'x'] [5, 11, 'x'] [5, 11, 'x']

140663007067328 140663007067328 140663007088192

We can see here that the variables *listone* and *listtwo* refer to the same location, but by using the slice operator and assigning the result to *listthree*, the *listthree* variable refers to a separate location. This does not happen with strings or tuples, however, because they are immutable.

*>>> strone = "hi"*

*>>> strtwo = strone*

*>>> strthree = strone[:]*

*>>> print(strone, strtwo, strthree)*

*>>> print(id(strone), id(strtwo), id(strthree))*

hi hi hi

140662877875888 140662877875888 140662877875888

**6.2.2 Shortcut Assignments**

Python has what are called ***shortcut assignment operators***. These add no power to the language, but save a little typing.

For example, the statement *num += 1* is equivalent to the statement *num = num + 1*.

*>>> num = 4*

*>>> num += 1*

*>>> num*

5

This also works for concatenation, e.g.:

*>>> str = 'hi'*

*>>> str += 'x'*

*>>> str*

'hix'

The shortcut is that instead of typing the name of the variable twice, once on the left and once on the right of the assignment operator =, we put the + to the left of the assignment operator and then only have to type the name of the variable once.

This also works for other operators, e.g. using \*= to multiply:

*>>> inumb = 3*

*>>> inumb \*= 2*

*>>> inumb*

6

This takes the current value of the variable *inumb*, multiplies by 2, and stores the result in a new location pointed to by the variable *inumb*.

Note that for mutable types, shortcut assignments modify the contents of the variable rather than creating a new variable.

*>>> list = [1, 5, 33]*

*>>> print(list, id(list))*

[1, 5, 33] 140437475761920

*>>> list += [14]*

*>>> print(list, id(list))*

[1, 5, 33, 14] 140437475761920

*>>> list = list + [29]*

*>>> print(list, id(list))*

[1, 5, 33, 14, 29] 140437475751424

**6.2.3 Simultaneous Assignments**

We have seen the simultaneous assignment statement, in which there are multiple variables on the left of the assignment operator, and an equal number of expressions on the right. For example, the following code initializes the variables *mysum* and *mycount* to have the value 0.

*>>> mysum, myproduct = 0, 1*

Simultaneous assignments can be used to exchange the values of two variables. The following code does not produce the desired result, since the assignment *a = b* will assign 10 to the variable *a*, and the original value of *a* (which was 3) is lost.

*>>> a = 3*

*>>> b = 10*

*>>> print(a,b)*

*>>> a = b*

*>>> b = a*

*>>> print(a,b)*

3 10

10 10

Generally in programming it is necessary to use a temporary variable for this.

*>>> a = 3*

*>>> b = 10*

*>>> print(a, b)*

*>>> temp = a*

*>>> a = b*

*>>> b = temp*

*>>> print(a,b)*

3 10

10 3

By storing the original value of *a* in the variable *temp*, the value is not lost and can then be assigned to *b*.

However, in Python this can be accomplished using the simultaneous assignment, since the assignments can be thought of as simultaneous and not sequential.

*>>> a, b = 3, 10*

*>>> print(a,b)*

*>>> a,b = b,a*

*>>> print(a,b)*

*3 10*

*10 3*

**6.2.4 Passing Arguments to Functions**

In Python, arguments are passed to functions using a methodology that is sometimes referred to as ***call-by-assignment***. That is, passing arguments to function parameters in Python works in the same way as assigning one variable to another variable. Passing an argument to a function parameter results in the function parameter referring to the same location as the argument.

The following example illustrates this for an argument which is an immutable type, in this case a number.

*def testcall(mynum):*

*""" Tests results of a function call with a number argument. """*

*print('Fn: ', mynum, id(mynum))*

*mynum = 14*

*print('Fn: ', mynum, id(mynum))*

*>>> num = 5*

*>>> print(num, id(num))*

*>>> print('Now function is called')*

*>>> testcall(num)*

*>>> print('Function has ended')*

*>>> print(num, id(num))*

5 140503077231024

Now function is called

Fn: 5 140503077231024

Fn: 14 140503077231312

Function has ended

5 140503077231024

In this example, the integer 5 was assigned to a variable *num*, and the value that *num* refers to and its id were printed. Then, the function was called. This causes the parameter *mynum* in the function to refer to the same location that *num* refers to. This can be seen from the result of the first **print** statement in the function. Then, a new value, 14, was assigned to *mynum* which means that *mynum* now refers to a new location in which the 14 is stored. The function ends at that point, and we can see that after the function has finished executing, the variable *num* has not changed.

Similar results will be obtained for other immutable parameter types, such as strings or tuples.

Now let’s try basically the same example, but using a list instead of a number. Since assignment statements always create a new location, the result is similar to the result when the argument is a number.

*def testcalllist(mylist):*

*""" Tests results of a fn call with a list argument. """*

*print('Fn: ', mylist, id(mylist))*

*mylist = [33, 5, 11]*

*print('Fn: ', mylist, id(mylist))*

*>>> listarg = [44, 2, -99]*

*>>> print(listarg, id(listarg))*

*>>> print('Now function is called')*

*>>> testcalllist(listarg)*

*>>> print('Function has ended')*

*>>> print(listarg, id(listarg))*

[44, 2, -99] 140503223213248

Now function is called

Fn: [44, 2, -99] 140503223213248

Fn: [33, 5, 11] 140503223315136

Function has ended

[44, 2, -99] 140503223213248

However, if the list parameter is modified within the function (instead of assigning a new list), the list argument that was passed to the function will change.

def testcalllistchg(mylist):

""" Tests results of a fn call with a list argument. """

print('Fn: ', mylist, id(mylist))

mylist.append(8)

print('Fn: ', mylist, id(mylist))

*>>> listarg = [44, 2, -99]*

*>>> print(listarg, id(listarg))*

*>>> print('Now function is called')*

*>>> testcalllistchg(listarg)*

*>>> print('Function has ended')*

*>>> print(listarg, id(listarg))*

[44, 2, -99] 140503223251712

Now function is called

Fn: [44, 2, -99] 140503223251712

Fn: [44, 2, -99, 8] 140503223251712

Function has ended

[44, 2, -99, 8] 140503223251712

In many cases, having the function modify the list variable is not a desired outcome. To protect the list from being changed, a slice consisting of the entire list can be passed instead of just the list variable:

*>>> listarg = [44, 2, -99]*

*>>> print(listarg, id(listarg))*

*>>> print('Now function is called')*

*>>> testcalllistchg(listarg[:])*

*>>> print('Function has ended')*

*>>> print(listarg, id(listarg))*

[44, 2, -99] 140503223187200

Now function is called

Fn: [44, 2, -99] 140503223252480

Fn: [44, 2, -99, 8] 140503223252480

Function has ended

[44, 2, -99] 140503223187200