Lecture 2.3 : Variables, references, immutable and mutable objects

Introduction ¶

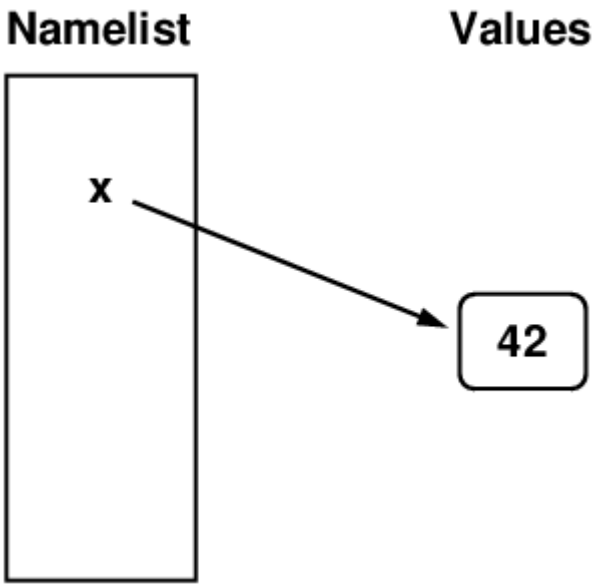
- A variable can contain a reference to an *immutable* object or a *mutable* object. We examine the different behaviours that arise when working with each.

Variables, references and immutable objects

- What exactly happens when Python executes the following statement?

```
>>> x = 42
```

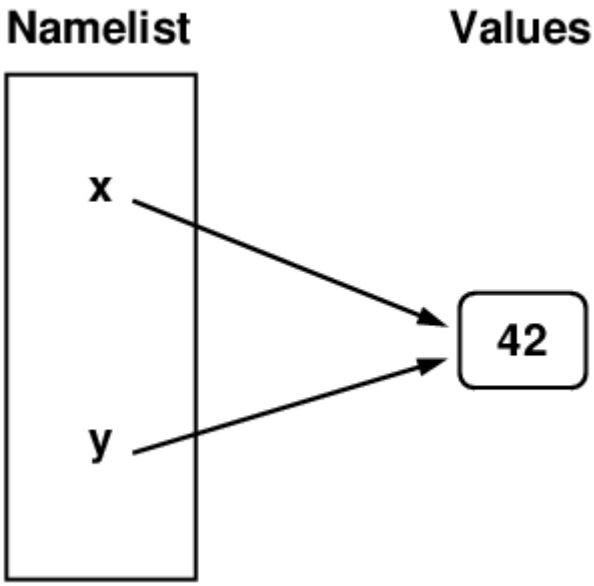
- A variable only comes into existence once it has been associated with a particular value. In the case above the variable `x` comes into existence and we can thenceforth refer to `x`. Python maintains a *namespace* of mappings from variables to the objects they refer to. After the above statement is executed a mapping from `x` to the value `42` is added to the namespace. This is depicted below.



- As we can see, the variable `x` *does not contain* the value `42`. Instead some memory is reserved to hold the value `42` and `x` *points to* that location in memory. We say that `x` contains a *reference* to the number `42` stored in memory.

- What happens the picture when we execute the following statements?

```
>>> x = 42
>>> y = x
```



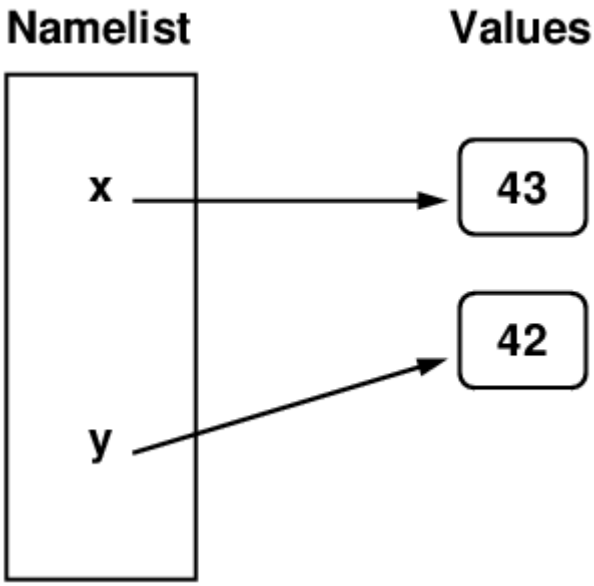
- We see that having executed `y = x` both `x` and `y` reference the same object in memory, in this case the integer `42`. We can verify this is the case by using the `id` function to print out the location in memory (i.e. memory address) that each of `x` and `y` reference:

```
>>> x = 42
>>> y = x
>>> id(x)
235481190240
>>> id(y)
235481190240
>>> id(x) == id(y)
True
```

- What happens the picture when we modify `x` as shown below?

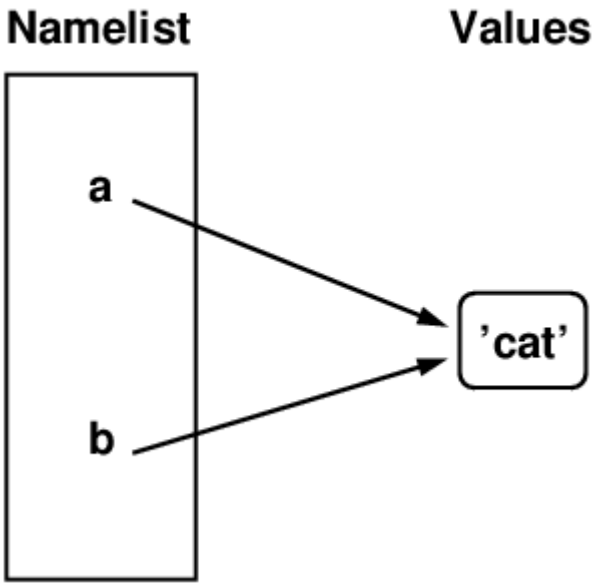
```
>>> x = 42
>>> y = x
>>> x += 1
>>> x
43
>>> y
42
```

- Executing `x += 1` is equivalent to executing `x = x + 1`. Evaluating the right hand side gives a new integer `43` and we set `x` pointing to it in memory. Thus the effect of executing `x += 1` is to *overwrite* `x` with a *new* reference to a *new* integer object. This time the reference is to the integer `43`. Note that overwriting the reference in `x` with a new one has *no effect* on the reference in `y`. It still points to `42`.



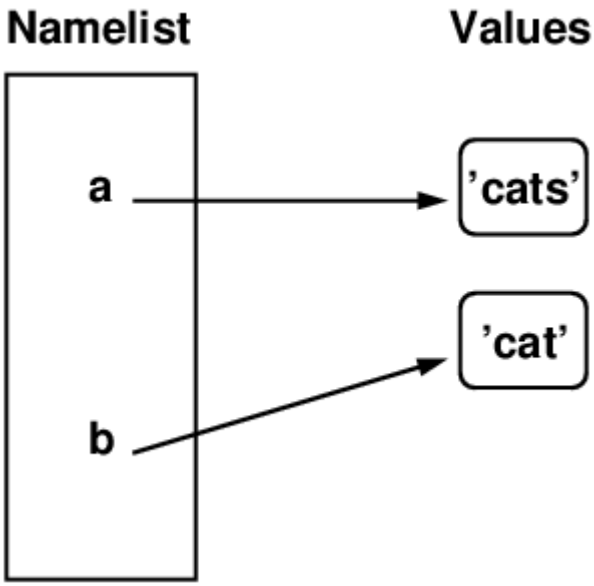
- The `int` type is *immutable*. This means operations on an integer create a *new* integer. Thus a modification of the integer referenced by `x` creates a *new* integer leaving the integer pointed to by `y` unchanged.
- Integers are not the only immutable type we have met. Strings are also immutable and behave similarly.

```
>>> a = 'cat'
>>> b = a
```



- If we modify the string referenced by `a` we get a new string. (Remember every operation on an *immutable* object creates a *new* object.) Below we overwrite `a` with a reference to this new string leaving `b` unchanged:

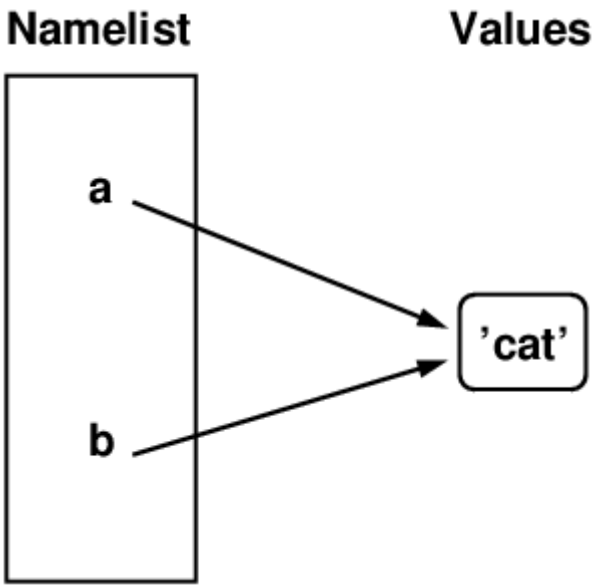
```
>>> a = 'cat'
>>> b = a
>>> a += 's'
>>> a
'cats'
>>> b
'cat'
```



Equality and identity

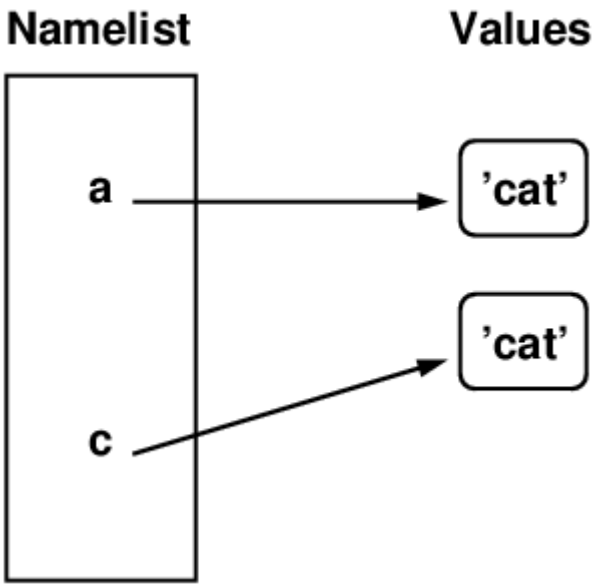
- We can check whether the objects referenced by two variables are equal using the `==` operator.
- We can check whether two variables reference the *same* object using the `is` operator.
- If `a is b` then `a == b`.
- If `a == b` it is not necessarily true that `a is b`.

```
>>> a = 'cat'
>>> b = a
>>> a == b
True
>>> a is b
True
>>> id(a)
139954150187504
>>> id(b)
139954150187504
>>> id(a) == id(b)
True
```



```
>>> a = 'cat'
>>> b = 'catastrophe'
>>> c = b[0:3]
>>> a
'cat'
>>> c
'cat'
>>> a == c
True
>>> a is c
False
>>> id(a)
139954150187504
>>> id(c)
139954150188512
>>> id(a) == id(c)
False
```

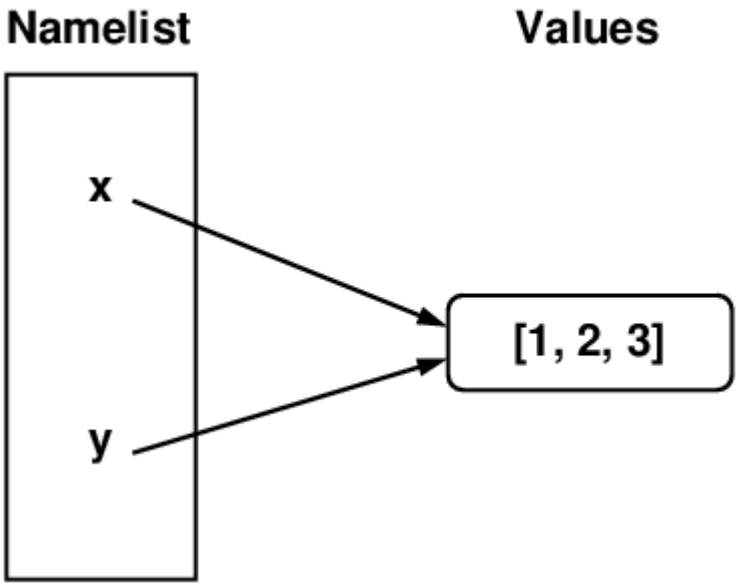
- Above we can see that both `a` and `c` point to `cat` but it is not the same `cat` in memory. Thus although they are equal i.e. `a == c` is `True` they do *not* reference the same object i.e. `a is c` is `False`.



Variables, references and mutable objects

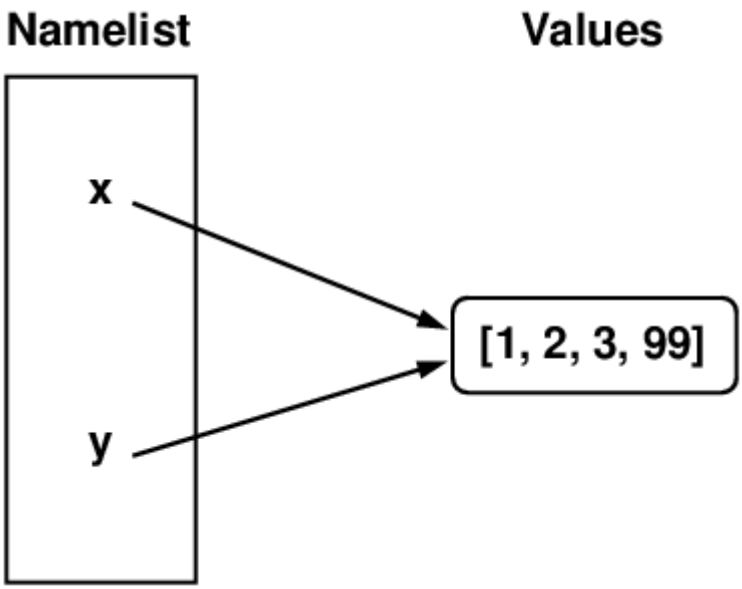
- Things get more complicated when multiple variables reference the same *mutable* object. We need to be careful as there are consequences to such sharing that may not be immediately obvious.

```
>>> x = [1, 2, 3]
>>> y = x
>>> x == y
True
>>> x is y
True
```



- Consider what happens to `y` when we write *through* the reference in `x` in order to append an element to the underlying list (and similarly for `x` if we write *through* the reference in `y`):

```
>>> x.append(99)
>>> x
[1, 2, 3, 99]
>>> y
[1, 2, 3, 99]
>>> x == y
True
>>> x is y
True
```



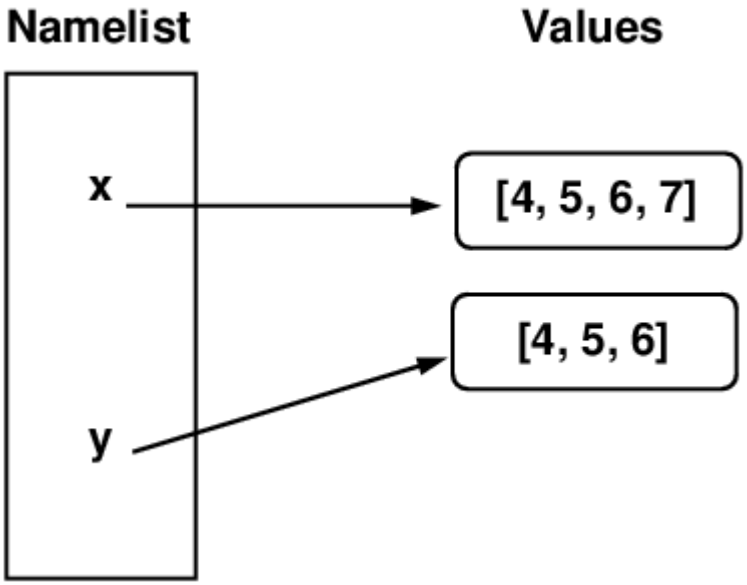
- The crucial point to note here is that because the object pointed to by `x` (and `y`) is *mutable*, modifying it does *not* create a new object and the original reference is *not* overwritten to point to a new object. We do not overwrite the reference in the variable `x` but instead we write *through* it to modify the mutable object it points to.

- There are however some gotchas with this behaviour. One is illustrated below.

```
>>> x = [4, 5, 6]
>>> y = x
>>> x = x + [7]
>>> x
[4, 5, 6, 7]
```

```
>>> y
[4, 5, 6]
```

- Huh? How come `y` was not modified in this example even though we made a change to `x`? Well here we executed the code `x = x + [7]`. However the critical difference is that the latter code does *not* write *through* `x` to modify the underlying list. Instead the evaluation of the right hand side (`x + [7]`) builds a *new* list made up of the concatenation of list `x` and list `[7]`. A reference to this *new* list overwrites the original reference in `x`. The list referenced by `y` is unchanged.

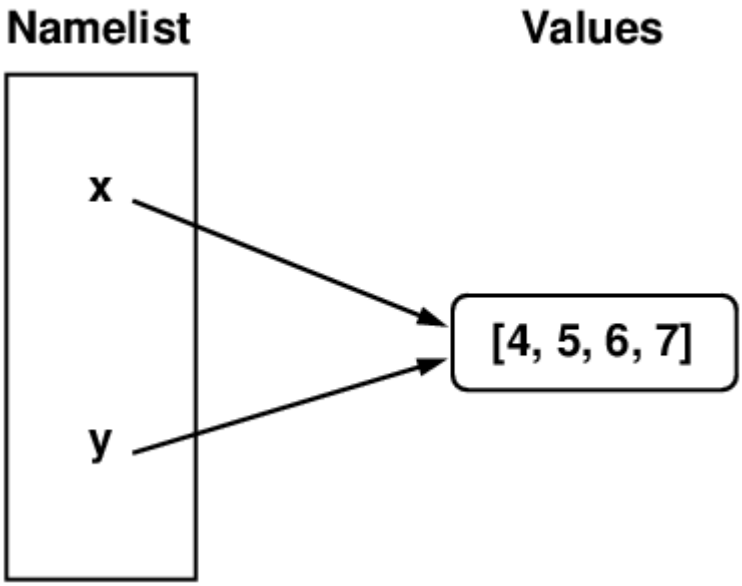


- We are not yet finished exploring the subtleties of this behaviour however. Consider the example below.

```
>>> x = [4, 5, 6]
>>> y = x
>>> x += [7]
>>> x
[4, 5, 6, 7]
>>> y
[4, 5, 6, 7]
```

- Huh? How come `y` was modified in this example? Given the preceding example we might expect `y` to be unchanged despite the change to `x` i.e. we might expect `x = x + [7]` (from the previous example) and `x += [7]` (from above) to be equivalent. However they are *not* equivalent and `y` is indeed changed.

- So what is going on? It turns out that the `+=` operator when applied to a list modifies the list *in-place*. This means we effectively write *through* the reference in `x` to append the contents of `[7]` to `x` when we write `x += 7`. Contrast this with where we *overwrite* `x` with a reference to a new list when we write `x = x + [7]`.



- In summary, an operation on an immutable object must create a new object. An operation on a mutable object *may* create a new object or modify in-place the underlying object. It depends on the particular operator's implementation.

How do I create a fresh copy of a list?

- As we have seen, to create a new copy of a list `x` we cannot simply write `y = x` since this only makes `y` an *alias* for `x` (i.e. they each reference the same object). So how can we create a *new* and *separate* copy of the list referenced by `x`? We can do so using the slice operator `:` as follows:

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> x
[1, 2, 3]
>>> y
[1, 2, 3]
>>> x == y
True
>>> x is y
False
```