Mutable vs Immutable (and Hashable) in Python

Python 2.x 2.2–2.7 , Python 3.x 3.0–3.6

Examples

Mutable and Immutable as Arguments

One of the major use case when a developer needs to take mutability into account is when passing arguments to a function. This is very important, because this will determine the ability for the function to modify objects that doesn't belong to its scope, or in other words if the function has side effects. This is also important to understand where the result of a function has to be made available.

```
>>> def list_add3(lin):
    lin += [3]
    return lin

>>> a = [1, 2, 3]
>>> b = list_add3(a)
>>> b
[1, 2, 3, 3]
>>> a
[1, 2, 3, 3]
```

Here, the mistake is to think that `lin`, as a parameter to the function, can be modified locally. Instead, `lin` and `a` reference the same object. As this object is mutable, the modification is done in-place, which means that the object referenced by both `lin` and `a` is modified. `lin` doesn't really need to be returned, because we already have a reference to this object in the form of `a`. `a` and `b` end referencing the same object.

This doesn't go the same for tuples.

```
>>> def tuple_add3(tin):
    tin += (3,)
    return tin

>>> a = (1, 2, 3)
>>> b = tuple_add3(a)
>>> b
(1, 2, 3, 3)
>>> a
(1, 2, 3)
```

At the beginning of the function, `tin` and `a` reference the same object. But this is an immutable object. So when the function tries to modify it, `tin` receive a new object with the modification, while `a` keeps a reference to the original object. In this case, returning `tin` is mandatory, or the new object would be lost.

Exercise

```
>>> def yoda(prologue, sentence):
    sentence.reverse()
    prologue += " ".join(sentence)
    return prologue

>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

Note: `reverse` operates in-place.

What do you think of this function? Does it have side effects? Is the return necessary? After the call, what is the value of `saying`? Of `focused`? What happens if the function is called again with the same parameters?

Mutable vs Immutable

There are two kind of types in Python. Immutable types and mutable types.

Immutables

An object of an immutable type cannot be changed. Any attempt to modify the object will result in a copy being created.

This category includes: integers, floats, complex, strings, bytes, tuples, ranges and frozensets.

To highlight this property, let's play with the `id` builtin. This function returns the unique identifier of the object passed as parameter. If the id is the same, this is the same object. If it changes, then this is another object. *(Some say that this is actually the memory address of the object, but beware of them, they are from the dark side of the force...)*

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

Okay, 1 is not 3... Breaking news... Maybe not. However, this behaviour is often forgotten when it comes to more complex types, especially strings.

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
140128123955504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

Aha! See? We can modify it!

```
>>> id(stack)
140128123911472
```

No. While it seems we can change the string named by the variable `stack`, what we actually do, is creating a new object to contain the result of the concatenation. We are fooled because in the process, the old object goes nowhere, so it is destroyed. In another situation, that would have been more obvious:

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

In this case it is clear that if we want to retain the first string, we need a copy. But is that so obvious for other types?

Exercise

Now, knowing how a immutable types work, what would you say with the below piece of code? Is it wise?

```
s = ""
for i in range(1, 1000):
    s += str(i)
    s += ","
```

Mutables

An object of a mutable type can be changed, and it is changed *in-situ*. No implicit copies are done.

This category includes: lists, dictionaries, bytearrays and sets.

Let's continue to play with our little `id` function.

```
>>> b = bytearray(b'Stack')
>>> b
bytearray(b'Stack')
>>> b = bytearray(b'Stack')
>>> id(b)
140128030688288
>>> b += b'Overflow'
>>> b
bytearray(b'StackOverflow')
>>> id(b)
140128030688288
```

(As a side note, I use bytes containing ascii data to make my point clear, but remember that bytes are not designed to hold textual data. May the force pardon me.)

What do we have? We create a bytearray, modify it and using the `id`, we can ensure that this is the same object, modified. Not a copy of it.

Of course, if an object is going to be modified often, a mutable type does a much better job than an immutable type. Unfortunately, the reality of this property is often forgotten when it hurts the most.

```
>>> c = b
>>> c += b' rocks!'
>>> c
bytearray(b'StackOverflow rocks!')
```

Okay...

```
>>> b
bytearray(b'StackOverflow rocks!')
```

Waiiit a second...

```
>>> id(c) == id(b)
True
```

Indeed. `c` is not a copy of `b`. `c` is `b`.

Exercise

Now you better understand what side effect is implied by a mutable type, can you explain what is going wrong in this example?

```
>>> ll = [ [] ]*4 # Create a list of 4 lists to contain our results
>>> ll
[[], [], [], []]
>>> ll[0].append(23) # Add result 23 to first list
>>> ll
[[23], [23], [23], [23]]
>>> # Oops...
```

Syntax

Parameters

Remarks