# PYTHON OBJECTS: MUTABLE VS. IMMUTABLE

December 24, 2013tgroshonmutability, python, types

Not all python objects are created equal. Some objects are mutable, meaning they can be altered, while others are immutable; pretty much the opposite of mutable🙂. So what does this mean for you when writing code in python? This post will talk about (a) the mutability of common data types and (b) instances where mutability matters.

## MUTABILITY OF COMMON TYPES

The way I like to remember which types are mutable and which are not is that *containers* and *user-defined* types are generally mutable while everything else is immutable. Then I just remember the exceptions like*tuple* which is an immutable container and *frozen set* which is an immutable version of *set* (which makes sense, so you just have to remember *tuple*).

The following are **immutable** objects:

- Numeric types: int, float, complex
- string
- tuple
- frozen set
- bytes

The following objects are **mutable**:

- list
- dict
- set
- byte array

What if I need a mutable string to do something like character swapping? Well then use a byte array.

## WHEN MUTABILITY MATTERS

Mutability might seem like an innocuous topic, but when writing an efficient program it is essential to understand. For instance, the following code is a straightforward solution to concatenate a string together:

```
1  string_build = ""
2  for data in container:
3      string_build += str(data)
4  print(string_build)
```

But in reality, it is extremely inefficient. Because strings are immutable, concatenating two strings together actually creates a third string which is a combination of the previous two. If you are iterating a lot and building a large string, you will waste a lot of memory creating and throwing away objects. Also, do not forget that nearing the end of the iteration you will be allocating and throwing away very large string objects which is even more costly.

The following is a more efficient and pythonic method:

```
builder_list = []
for data in container:
    builder_list.append(str(data))
"".join(builder_list)

### My favorite way is to use a list comprehension
### which is cleaner code and runs faster

"".join([str(data) for data in container])
```

This code takes advantage of the mutability of a single list object to gather your data together and then allocate a single result string to put your data in. That cuts down on the total number of objects allocated by almost half.

Another gotcha related to mutability is this little number:

```
def my_function(param=[]):
    param.append("thing")
    return param
```

What you might think would happen is that by giving an empty list as a default value to `param`, a new empty list is allocated each time the function is called and no list is passed in. But what actually happens is that because Python only evaluates functions definitions once and a list is a mutable object, every call that uses the default list **will be using the same list**

```
print(my_function())  # prints: ["thing"]
print(my_function())  # prints: ["thing", "thing"]
```

Do not put a mutable object as the default value of a function parameter. Immutable types are perfectly safe. If you want to get the intended effect, do this.

```
def my_function2(param=None):
    if param is None:
        param = []
    param.append("thing")
    return param
```

# CONCLUSION

Mutability matters. Learn it. Primitive-like types are probably immutable. Container-like types are probably mutable.