Mutable and Immutable Objects

Definition

> **Mutable**: Liable to change. **Mutable** objects can be changed.

> **Immutable**: Unchanging over time or unable to be changed. **Immutable** objects cannot be changed.

> In Python, everything is an **object**, and every object *has* (or *is of*) a **data type**, and depending on the data type, an object can either be **mutable** or **immutable**.

Mutable & Immutable

A list is **mutable**; it can have items/values added, removed, or changed. A string is **immutable**; you cannot have its characters removed or changed, and you cannot add more characters to it. If you want to remove some characters in a string, you have to create a new string without those characters.

If we try to modify a single character in a string **in-place**, we will encounter a `TypeError` exception:

```
>>> name = "King is a cat"
>>> name[8] = "the"
Traceback (most recent call last):
    File "<pyshell#50>", line 1, in <module>
        name[8] = 'the'
TypeError: 'str' object does not support item assignment
```

The proper way to **"mutate"** a string is to use *slicing* and *concatenation* to build a **new** string by copying from the parts of the old string:

```
>>> name = "King is a cat"
>>> new_name = name[0:5] + "the" + name[9:13]
>>> name
'King is a cat'
>>> new_name
'King the cat'
```

As you can see from the above example, we used `[0:5]` and `[9:13]` to refer to the characters that we wanted to keep, and notice that the original "King is a cat" string is not modified because a string is **immutable**.

Mutable Objects

Mutable objects in Python are: lists, dictionaries, and sets. The values that are contained in lists, dictionaries, and sets can be changed **in-place** without having to create a separate copy of them with the changes. Now, we will have a look at some examples of a list object.

> In Python, **mutable** objects are those whose values **can** be changed **in-place** after assignment.

List (mutable)

Although a list is mutable, the second line in the following code does not modify the list what is stored in `eggs`:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```

The list `[1, 2, 3]` which is stored in the `eggs` variable is not being changed here; rather, an entirely new and different list `[4, 5, 6]` gets referenced by the `eggs` variable; you're not actually modifying the values of the list `[1, 2, 3]` itself.

If you wanted to actually modify the original list in `eggs`, you would have to do something like the following:

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

Or you can do it like this:

```
>>> eggs = [1, 2, 3]
>>> eggs[0] = 4
>>> eggs[1] = 5
>>> eggs[2] = 6
>>> eggs
[4, 5, 6]
```

Immutable Objects

Immutable objects in Python are: integers, floats, strings, and tuples. Once you create a variable which has a value that is immutable, you cannot change the value **in-place**.

> In Python, **immutable** objects are those whose values **cannot** be changed **in-place** after assignment.

Strings (immutable)

List is not the only thing that represents an ordered sequence of values. For example, strings and lists are actually very similar if you consider a string to be a "list" of characters. Many of the things you can do with lists can also be done with strings: indexing, slicing, and using them with loops, using `len()` function to get the length of the string, and `in` and the `not in` operators to see if some characters exist in the string.

```
>>> name = "abc"
>>> name[0]
'a'
>>> name[-2]
'i'
>>> name[0:3] # slicing
'abc'
>>> "a" in name
True
>>> "z" in name
False
>>> for i in name:
        print i
a
b
c
```

Tuple (immutable)

The `tuple` data type is almost identical to the list data type, except in two ways. First, tuples are written with parentheses, `(` and `)` instead of square brackets `[` and `]`.

```
>>> eggs = ("hello", 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

However, tuples are different from lists in that, tuples, like strings, are **immutable**. Tuples cannot have their values modified, appended, or removed **in-place**.

Here is an example that attempts to modify a value in a tuple.

```
>>> eggs = ("hello", 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
    File "<pyshell#5>", line 1, in <module>
        eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

Benefits of Tuples

You can use tuples to convey to anyone reading your code that you do not intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple.

A second benefit of using tuples instead of lists is that, because they are immutable and their contents do not change, Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Mutable & Immutable Objects as Function Arguments

> When a variable (which has a mutable object) is passed as an argument to a function, the reference to the object is passed and the parameter of the function will then have the reference to the object.

> When a variable (which has an immutable object) is passed as an argument to a function, a complete copy of the value is created and assigned to the parameter of the function.