6.3 Mutable Data Types

In the previous section, we introduced the concept of **object mutation**, and saw how we could mutate Python lists with the `list.append` method. In this section, we'll survey some of the other ways of mutating lists, and see what other data types can be mutated as well. For a full reference of Python's mutating methods on these data types, please see Appendix A.2 Python Built-In Data Types Reference.

Mutable and immutable data types

As we saw in the last section, the Python interpreter stores data in entities called *objects*, where each object has three fundamental components: its id, type, and value. The data type of an object determines what the allowed values of that object are (e.g., an `int` object can have value `3` but not `'David'`), as well as what operations can be performed on that object (e.g., what `+` means for `int`s vs. `str`s). One consequence of the latter point is that an object's data type determines whether any mutating operations can be performed on the object—in other words, it is the object's data type that determines whether it can be mutated or not.

We say that a Python data type is **mutable** when it supports at least one kind of mutating operation, and **immutable** when it does not support any mutating operations. So which data types are mutable and immutable in Python?

- The non-collection data types `int`, `float`, `bool`, `str` are all *immutable*.
- The collection data types `set`, `list`, and `dict` are all *mutable*.
- Data classes are *mutable* by default.[1]

[1] There is a way to define immutable data classes, but that is beyond the scope of this course.

Instances of an immutable data type cannot change their value during the execution of a Python program. So for example, if we have an `int` object with value `3`, that object's value will *always* be `3`. But remember, a variable that refers to this object might be reassigned to a different object later. This is why is is important that we differentiate between variables and objects!

Comment: why immutability?

By definition, mutable data types are more flexible than immutable data types—they can do something that immutable data types cannot. So you might wonder why Python has immutable data satypes at all, or put another, why can't we just mutate any object?

As we'll discuss later in the course, in software design there is almost always a trade-off between the *functionality* provided by software and the *code complexity and efficiency* of the implementation of that software. Intuitively, the more kinds of operations that a given data type (or more generally, programming language) supports, the more code that needs to be written to implement those operations, and the more flexible the underlying data representations need to be. By choosing to make some data types immutable, the Python programming language designers are then able to simplify the code for handling those data types in the Python interpreter, and in doing so make the remaining non-mutating operations less error-prone and more efficient. The price for this that we pay as Python programmers is that it is our responsibility to keep track of which data types are mutable and which ones aren't.

Example: `list`s and `tuple`s

All the way back in 1.7 Building Up Data with Comprehensions, we mentioned briefly that there was a Python data type, `tuple`, that was similar to `list` and that could also be used to represent sequences. So far, we've been treating `tuple`s interchangeably with `list`s.

Now that we've discussed mutability, we are ready to state the difference between `list` and `tuple`: *in Python, a `list` is mutable, and a `tuple` is immutable.* For example, we can modify a `list` value by adding an element with `list.append`, but there is no equivalent `tuple.append`, nor any other mutating method on tuples.

Mutating `list`s

For the remainder of this section, we'll briefly describe the various mutating operations we can perform on the mutable data types we've seen so far in this course. Let's start with `list`.

`list.append`, `list.insert`, and `list.extend`

In addition to `list.append`, here are two other methods that adding new elements to a Python list. The first is `list.insert`, which takes a list, an *index*, and an object, and inserts the object at the given index into the list.

```
>>> strings = ['a', 'b', 'c', 'd']
>>> list.insert(strings, 2, 'hello')  # Insert 'hello' into strings at index 2
>>> strings
['a', 'b', 'hello', 'c', 'd']
```

The second is `list.extend`, which takes two lists and adds all elements from the second list at the end of the first list, as if `append` were called once per element of the second list.

```
>>> strings = ['a', 'b', 'c', 'd']
>>> list.extend(strings, ['CSC110', 'CSC111'])
>>> strings
['a', 'b', 'c', 'd', 'CSC110', 'CSC111']
```

List index assignment

There is one more way to put a value into a list: by overwriting the element stored at a specific index. Given a list `lst`, we've seen that we can access specific elements using indexing syntax `lst[0]`, `lst[1]`, `lst[2]`, etc. We can also use this kind of expression as the *left side* of an assignment statement to mutate the list by modifying a specific index.

```
>>> strings = ['a', 'b', 'c', 'd']
>>> strings[2] = 'Hello'
>>> strings
['a', 'b', 'Hello', 'd']
```

Note that unlike `list.insert`, assigning to an index removes the element previously stored at that index from the list!

List augmented assignment

And now let us return to *augmented assignment statements* that we first introduced in 6.1 Variable Reassignment, Revisited. You already know that Python `list`s support concatenation using the `+` operator; now let's see what happens when we use a `list` on the left-hand side of a `+=` augmented assignment statement:

```
>>> strings = ['a', 'b', 'c', 'd']
>>> strings += ['Hello', 'Goodbye']
>>> strings
['a', 'b', 'c', 'd', 'Hello', 'Goodbye']
```

So far, this seems to fit the behaviour we saw for numbers: `strings += ['Hello', 'Goodbye']` looks like it does the same thing as `strings = strings + ['Hello', 'Goodbye']`. But it doesn't! Let's look at ids to verify that it doesn't.

```
>>> strings = ['a', 'b', 'c', 'd']
>>> id(strings)
1920488009536
>>> strings += ['Hello', 'Goodbye']
>>> strings
['a', 'b', 'c', 'd', 'Hello', 'Goodbye']
>>> id(strings)
1920488009536
```

After the augmented assignment statement, *the id of the object that `strings` hasn't changed*. This means that the variable `strings` wasn't actually reassigned, but instead the original list object was mutated instead. In other words, for lists `+=` behaves like `list.extend`, and *not* like "`x = x + 3`". This may seem like inconsistent behaviour, but again the Python programming language designers had a purpose in mind: they wanted to encourage object mutation rather than variable reassignment for this list operation, because the former is more efficient when adding new items to a list.[2]

[2] This is precisely the same reasoning we used when comparing our two versions of `squares` from the previous section.

Mutating `set`s

Python `set`s are mutable. Because they do not keep track of order among the elements, they are simpler than `list`s, and offer just two main mutating methods: `set.add` and `set.remove`, which (as you can probably guess) add and remove an element from a set, respectively.[3] We'll illustrate `set.add` by showing how to re-implement our `squares` function from the previous section with `set` instead of `list`:

[3] The `list` data type also provides a few mutating methods that remove elements, though we did not cover them in this section.

```
def squares(numbers: set[int]) -> set[int]:
    """Return a set containing the squares of all the give

    ...
    """
    squares_so_far = set()
    for n in numbers:
        set.add(squares_so_far, n * n)

    return squares_so_far
```

Note that `set.add` will only add the element if the set does not already contain it, as sets cannot contain duplicates. In addition, `list.append` will add the element to the end of the sequence, whereas `set.add` does not specify a "position" to add the element.

Mutating dictionaries

The most common ways for dictionaries to be mutated is by adding a new key-value pair or changing the associated value for a key-value pair in the dictionary. This does not use a `dict` method, but rather the same syntax as assigning by list index.

```
>>> items = {'a': 1, 'b': 2}
>>> items['c'] = 3
>>> items
{'a': 1, 'b': 2, 'c': 3}
```

The second assignment statement adds a new key-value pair to `items`, with the key being `'c'` and the items being `3`. In this case, the left-hand side of the assignment is not a variable but instead an expression representing a component of `items`, in this case the key `'c'` in the dictionary. When this assignment statement is evaluated, the right-hand side value `3` is stored in the dictionary items as the corresponding value for `'c'`.

Assignment statements in this form can also be used to mutate the dictionary by taking an existing key-value pair and replacing the value with a different one. Here's an example of that:

```
>>> items['a'] = 100
>>> items
{'a': 100, 'b': 2, 'c': 3}
```

Mutating data classes

As we said at the start of this section, Python data classes are mutable by default. To illustrate this, we'll return to our `Person` class:

```
@dataclass
class Person:
    """A person with some basic demographic information.

    Representation Invariants:
      - self.age >= 0
    """
    given_name: str
    family_name: str
    age: int
    address: str
```

We mutate instances of data classes by modifying their attributes. We do this by assigning to their attributes directly, using *dot notation* on the left side of an assignment statement.

```
>>> p = Person('David', 'Liu', 100, '40 St. George Street')
>>> p.age = 200
>>> p
Person(given_name='David', family_name='Liu', age=200, address='40 St. George Street')
```

One note of caution here: as you start mutating data class instances, you must always remember to respect the representation invariants associated with that data class. For example, setting `p.age = -1` would violate the `Person` representation invariant. To protect against this, `python_ta` checks representation invariants whenever you assign to attributes of data classes, as long as the `python_ta.contracts.check_contracts` decorator has been added to the data class definition.[4]

[4] See 5.3 Defining Our Own Data Types, Part 2 for a review on using `python_ta` to check representation invariants for a data class.