Chapter 15

Mutation

“All problems in computer science can be solved by another level of indirection.”
— Butler Lampson

“Mutants are not the ones mankind should fear.”
— Dr. Jean Grey, X-Men the Movie

“When you look at a lake, there is no water in your mind.”

Imagine that the town where you live has decided to invite a visitor from a far-off, remote tribal land, which still has a hunter-gatherer society. Missionaries taught the visitor, named Mzlot’l, how to speak English, but he has no experience with modern civilization. When he arrives, the Mayor throws a big party with lots of guests, who present Mzlot’l some gifts. He is amazed and delighted with some of these gifts, particularly enjoying a flashlight and a Swiss Army Knife, which he has to be shown how to open and use. However, when he is given a giftcard for $100, nobody can explain why this is a good gift to him—he finds this gift to be worthless and stupid. He cannot understand “you can use it to buy stuff” because he has no experience with stores and transactions. As far as Mzlot’l is concerned, a good gift is something that has immediate, visible function and worth. The giftcard is an abstract concept he does not fathom.

Money is a good example of an abstraction we use in everyday life. Money is the abstraction of value, especially paper money which has very little intrinsic worth (it’s just paper). Abstraction has merits and demerits. On the one hand, abstraction gives the freedom to substitute and manipulate quantities in many places. You can carry large amounts of money more easily than carrying raw goods, crops, lumber or gold. On the other hand, to get something useful out of money you need to exchange it for what is actually useful. In that sense, money is an indirect marker of value, requiring the user to do something to exploit it. Also, you might need to have some concern about whether or not the money is valid, i.e., it is not counterfeit or worthless because of some government collapse.

Just as money is an abstraction and has some indirect character, it will be seen that some Python variables are based on indirect mechanisms. Largely, this
feature of Python is hidden, and programmers may be unaware that variables may have references to values, rather than having values directly. This chapter exposes the use of indirection in certain Python data types, variables, methods, and operators.

The data types in Python fall into two categories, mutable and immutable. The mutable category includes lists, dictionaries, and sets. The other types introduced in Chapter 4 are all immutable types: bool, int, float, tuple, and string. It is the mutable types that use indirection in Python. These types are the subject of this chapter. Of the mutable types, the chapter concentrates on lists; working with dictionaries or sets has a similar flavor. The standard terms mutable and immutable are part of Python jargon, and used in the official reference manual for the language. In this chapter, the word mutation is used not to mean some biological, evolutionary process, but rather to form and reform data, like sculpting clay.

The technical matter of this chapter can be difficult to understand quickly. Different readers of this may comprehend some parts, but not others. The approach may seem tedious because the same idea is expressed in several ways and with many examples. Probably the most useful programming technique is slice assignment. The notion of indirection, as implemented for lists and dictionaries in Python, is a core idea in modern software. A patient reading of this chapter will give a solid foundation for object-oriented software design.

**Mutation vs Assignment**

Recall that there are two ways to change a variable representing a list, either assignment to the variable or indexed assignment (and deletion), which specifies an item as the target of the assignment. There is significant, if invisible, difference between these two.

```python
>>> A = [0,1,2,3,4]
>>> print("A = ",A)
A = [0, 1, 2, 3, 4]
>>> A = "time after time",split()
>>> print("A = ",A)
A = ['time', 'after', 'time']
```

Both assignments above give variable A a reference to a list; the first assignment creates A, and the second assignment replaces A’s reference, so that A refers to a new list. By contrast,

```python
>>> print("A = ",A)
['time', 'after', False]
```
The assignment here does not change A’s reference. Rather, it *mutates* the list referenced by A. These words may just seem like semantics, making some distinction that really has no practical consequence. Hold on. Later in the chapter, there are examples that will demonstrate these differences are important. For now, just be aware that there is some internal distinction between assigning without an index and assigning to an item.

**Slice Assignment**

An attractive syntax feature of Python is the ability to assign to a slice of a list. The syntax is simple, once the concept of slices is understood: specify a slice on the left side of “=” and that slice is replaced by the expression on the right side.

```python
>>> X = [2,4,6,8,10,12,14,16]
>>> X[2:2+3] = ['a','b','c']
>>> X
[2, 4, 'a', 'b', 'c', 12, 14, 16]
```

Note that the expression on the right side must be a sequence: since a slice itself is a sequence, Python needs to have a sequence to replace it. For example:

```python
>>> X = [2,4,6,8,10,12,14,16]
>>> X[2:3] = False
TypeError: can only assign an iterable
```

(consider the term *iterable*, for now, to be a sequence). So, even if the slice only has a single item, it is still a *list* of one item – a sequence is needed to replace it.

```python
>>> X = [2,4,6,8,10,12,14,16]
>>> X[2:3] = 'x'
>>> X
[2, 4, 'x', 8, 10, 12, 14, 16]
```

The above does not cause an error because 'x' is a sequence (strings are sequences). Types matter.

What makes slice assignment particularly useful is that the length of the slice and the length of the sequence on the right side of the assignment can differ.

```python
>>> X = [2,4,6,8,10,12,14,16]
>>> X[4:] = "1/2/3",split('/')
>>> X
[2, 4, 6, 8, '1', '2', '3']
```
Hence, slice assignment can delete items.

```python
>>> X = [2, 4, 6, 8, 10, 12, 14, 16]
>>> X[3:3+2] = []
[2, 4, 6, 12, 14, 16]
```

Or, slice assignment can be used to insert items, by replacing an empty slice.

```python
>>> X = [2, 4, 6, 8, 10, 12, 14, 16]
>>> X[2:2] = [True, False]
>>> X
[2, 4, True, False, 6, 8, 10, 12, 14, 16]
>>> X[0:0] = [100]
[100, 2, 4, True, False, 6, 8, 10, 12, 14, 16]
```

Slice assignment can even replace the entire list.

```python
>>> X = [2, 4, 6, 8, 10, 12, 14, 16]
>>> X[0:len(X)] = [1, 3, 5]
>>> X
[1, 3, 5]
```

This last example is another illustration of the two kinds of assignment, variable replacement and mutation. The last assignment is mutation, because it replaces the list that `X` refers to, whereas the assignment just before gives `X` a new reference to a list.

**Slice Deletion** The `del` statement can remove a range of items, as given by the slice notation:

```python
>>> L = [5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> del L[3:6]
>>> L
[5, 6, 7, 11, 12, 13]
>>> del L[2:2]
>>> L
[5, 6, 7, 11, 12, 13]
```

Observe that deleting an empty slice has no effect.
Mutating Methods

A number of sequence methods are *mutating methods*, which change the list referenced by a variable.

**sort** The `sort` method rearranges the items of a list into increasing order. A quick example of this:

```python
>>> Y = [5,2,9,3,1,0,8]
>>> Y.sort()
>>> Y
[0,1,2,3,5,8,9]
```

It’s possible to put keyword parameters to `sort` and get a decreasing-order sort. Notice that, unlike methods like `index` and `count`, the `sort` method returns `None` — that’s why you don’t see any output above after `Y.sort()`. Rather, `sort` quietly does its job, which is to mutate the list into a sorted order.

*Remark.* Python also has a built-in function `sorted` which takes a sequence argument and returns a sorted copy of the sequence; function `sorted` is non-mutating, whereas the `sort` method is a mutator.

**reverse** The `reverse` method mutates a list by putting the items in reverse order.

```python
>>> Y = "let us go there".split()
>>> Y.reverse()
>>> Y
['there', 'go', 'us', 'let']
```

(There’s also a non-mutating, built-in function named `reversed`.)

**append** The `append` method mutates a list by adding an item to the list.

```python
>>> Z = [5==5, 2>5, "RA", 7]
>>> Z.append(20)
>>> Z
[True, False, 'RA', 7, 20]
>>> Z.append([30])
>>> Z
[True, False, 'RA', 7, 20, [30]]
```

Note that the argument for `append` is an item, which does not need to be a sequence, The last part of the example makes this clear, as it adds a list as the last item.

*(others)* The Python reference manual describes several other mutating methods: `extend`, `insert`, `remove`, `pop`. These are not essential to know.
Mutation in Functions

The previous section on mutating methods presented a few methods and showed examples, but did not really explain how these methods can mutate a variable of type `list`. To get some idea of the programming techniques that a mutation method might use, this section shows how to write a mutating function. The interesting twist is how Python uses indirection to get around the limitation of variable scope. Mutating assignment can change the list of a variable of a function’s caller, without using the `global` statement.

```python
def NewAppend(L,x):
    L[len(L):] = [x]
...
>>> V = [1,3,4]
>>> NewAppend(V,99)
>>> V
[1,3,4,99]
```

Obviously from this example, a function is able to change a list given as a parameter. Does this contradict what Chapter 14 says about variable scope? The answer is no, but to understand why, we need to look a bit deeper at how Python does binding. When Python evaluates the function application `NewAppend(V,99)`, it creates a local dictionary for `NewAppend`, and puts two entries into this dictionary, one for `L` and one for `x`. The subtle point is that the value for `L` is not the list `[1,3,4]`; rather, it is a reference to that list. After the local dictionary has been created, but before the evaluation of the `NewAppend` completes, there are temporarily two variables that refer to the same list, `L` in `NewAppend`'s scope, and `V` in the interactive session. When `NewAppend` does a mutating assignment, the list referenced by both variables changes. Ways that a function can mutate a list are index assignment, slice assignment, or the use of a list mutating method.

To reinforce the point about mutating assignment in a function, here is a small change to the example above.

```python
def NewAppend(L,x):
    L[len(L):] = [x]
    L = [True]
...
>>> V = [1,3,4]
>>> NewAppend(V,99)
>>> V
[1,3,4,99]
```

There are now two assignments in `NewAppend`. The first one mutates `L`. The second is a non-mutating assignment, so it gives `L` a reference to a new list.
This reference to a new list does not change what \( V \) refers to, so the mutation to \( L \) is retained and accessible through \( V \), whereas the second assignment has no effect on \( V \). What happens if we reverse the order?

```python
def NewAppend(L, x):
    L = [True]
    L[len(L):] = [x]
...
>>> V = [1, 3, 4]
>>> NewAppend(V, 99)
>>> V
[1, 3, 4]
```

This time, before the mutating assignment was done, the reference of \( L \) had already been changed to something different from what \( V \) refers to.

## The is Operator

The previous examples and the discussion of mutation are some evidence that Python uses an indirect way of connecting variables to list values. The binding of a list variable to a parameter connects two variables to the same list, but there is a much simpler way to do this in Python.

```python
>>> K = [6, 2, 9, 4]
>>> Kcopy = K
>>> K[1] = True
>>> K
[6, True, 9, 4]
>>> Kcopy
[6, True, 9, 4]
```

The assignment \( Kcopy = K \) created variable \( Kcopy \) and gave it, as a value, a reference to the same list that \( K \) refers to. Therefore, when the list itself was mutated by index assignment, both \( K \) and \( Kcopy \) refer to the same, mutated list. Another way to see the distinction is to use more than one list and the `==` comparison operator.

```python
>>> J = [9, 9, 4]
>>> Jcopy = J
>>> W = [9, 9, 4]
>>> J == Jcopy, J == W, Jcopy == W
True, True, True
>>> Jcopy[2] = 'a'
>>> J == Jcopy, J == W, Jcopy == W
False, True, False
```
Two facts underly the example. First, even though the initial assignments to \( J \) and \( W \) give the same list value to each variable, it is clear from the later behavior that \( J \) and \( W \) to not refer to the *same* list. That’s made clear when the \( J \text{copy} \) indexed assignment mutates the list that \( J \) references. Although \( J \) and \( W \) refer to different lists, the first comparison \( J == W \) evaluates to *True*, because the two lists have identical items. One way to think about this example is to imagine Python has some table of names and values, like a dictionary.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( J )</td>
<td>( \alpha )</td>
</tr>
<tr>
<td>( J \text{copy} )</td>
<td>( \alpha )</td>
</tr>
<tr>
<td>( \beta )</td>
<td>( \beta )</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>([9, 9, 4])</td>
</tr>
<tr>
<td>( \beta )</td>
<td>([9, 9, 4])</td>
</tr>
</tbody>
</table>

The table contains some hidden names, known only to Python, which are \( \alpha \) and \( \beta \). These hidden names have lists as their values, but the variable names do not. The variable names have hidden names for values. This is why we say mutable variables are *indirect* references to values. Though it is more difficult to understand mutable than immutable types, the features of mutable variables are quite useful: there are mutating methods and ways to change lists passed as arguments to functions.

The concept of having multiple references to the same thing is an abstraction used widely in software. Probably the most familiar instance is how files and directories (folders) are organized. In Windows, you can create a *shortcut* to a file or application. A shortcut is not a copy of a file, because a copy would take up disk space; a shortcut is another name that can be used in place of the file, and moved to the desktop or some other folder. The Linux/Unix term for this is a *link* to a file. The Python jargon for two variables that refer to the same list is *alias*. For the example above, \( J \text{copy} \) is an alias for the list that \( J \) references.

**Equality of Reference**  Two list variables can be equal, when compared by the \( == \) operator, even when they refer to different lists. Is there some way to see if two variables refer to the same list without mutating one of them, then checking that the this mutation happened to the other one? Python has an operator just for this question.

```python
>>> E = F = "tick tock clock".split()
>>> G = "tick tock clock".split()
>>> E == F, E == G, F == G
(1, True, True)
```
The `is` operator compares the *references* for two variables. It returns `True` only when the two variables refer to the same list. Above, `E is F` remains `True` up to the statement `F = range(10)`, which is a non-mutating assignment. The assignment to `F` gives it a new reference, different from `E`'s.

### Augmented Assignment

The general rule for assignment, with regard to mutation, is that any kind of assignment except indexed or sliced assignment makes a new reference rather than mutating a list. It does not matter whether augmented assignment (*e.g.*, `+=`) or tuple assignment is used, what matters is whether the variable on the left side of the assignment is indexed or a slice. There is one exception to the general rule.

Recall that `append` is a mutating method that adds an item at the end of a list. A similar method, `extend`, inserts a list at the end of a list:

```python
>>> A = B = [1, 2]
>>> A.extend([3, 4])
>>> B
[1, 2, 3, 4]
```

The `extend` method is thus a kind of concatenation method. The exception to the general rule is that Python translates `“+=”` for lists into a method call to `extend`.

```python
>>> A = B = [1, 2]
>>> A += [3, 4]
>>> A is B
True
>>> B
[1, 2, 3, 4]
```

Python doesn’t use `extend` for what seems to be an equivalent way of concatenating.
Mutation

>>> A = B = [1,2]
>>> A = A + [3,4]
>>> A is B
False
>>> B
[1, 2]
>>> A
[1, 2, 3, 4]

The second assignment to \(A\) gave a new reference to \(A\) rather than mutate the list of the existing reference.

Cloning

When you use a copy machine to reproduce a printed sheet of paper, you get a new sheet of paper, hopefully identical to the original for practical purposes. You can tear the copy, write on it, crumple it, and the original is unaffected. We have seen that \(B = A\) does not make a copy of a list, it makes another reference to a list. A nice illustration of this fact is the following.

>>> A = [1,2]
>>> B = A
>>> del A
>>> B
[1, 2]

After \(B = A\), both variables refer to the same list. When variable \(A\) is deleted, the underlying list is not destroyed; there remains a reference \(B\) to the list. Python only destroys an actual list (and recycles memory) once all references to that list are gone.

Sometimes, making another reference to a list is exactly what is wanted. Parameter binding to list arguments makes another reference to a list, which is why functions can mutate list variables given in their arguments. In other situations, we may not want to make another reference, because a copy works better. A copy can be changed without changing the original. The easy way to make a copy is the following trick.

>>> A = [1,2]
>>> B = A[:]
>>> B == A
True
>>> B is A
False
Optional: Shallow vs Deep Copy  One of the tricky topics this chapter has not covered is the issue of indirect references to lists when they are nested. Nested lists can be copied in several ways, with the extremes being shallow copy and deep copy. A shallow copy is simply $B = A[:]$. What more could one ask of copying?

```python
>>> A = [1, [2, 3], 4]
>>> B = A[:]
>>> B[0] = 'x'
>>> B
['x', [2, 3], 4]
>>> A
[1, [2, 3], 4]
>>> B[1][0] = 'y'
>>> B
['x', ['y', 3], 4]
>>> A
[1, ['y', 3], 4]
```

The example shows that although $B$ is different from $A$, the second item of $B$ is actually a reference to the same list that the second item of $A$ references. A shallow copy only copies “one level” of a nested list. A deep copy would make copies at all levels of nesting. How to do a deep copy of a list is an advanced topic beyond this chapter.

**Exercises**

Most of these exercises use the Python2 style of the `print` statement and ask what will be printed.

1. What will be the result of this interaction with Python?

   ```python
   >>> M, varX, varY, t = ('team', [], 'sample', [True, False])
   >>> print len(M) * len(varX) + len(varY) * len(t)
   ```
>>> if = "el"
>>> if += "if"
>>> print if

(3) What is printed in this interactive session?

>>> t3,t4 = [50,2], "going going gone"
>>> t4 = t4.split()
>>> t3 += t4
>>> print t3

(4) What does Python print?

>>> a = b = "sub urban"
>>> a[3] = "-"
>>> print b

(5) For this question, first a function definition is given, then the interactive part follows. What does Python print?

def YR(z):
    z[0] *= 2

>>> X = Y = ["F","a","s","t"]
>>> YR(Y)
>>> print ".join(X+Y)

(6) What will Python print?

>>> A1 = {"A":9, "B":8, "C":7, "D":6}
>>> A2 = A1
>>> A2[0] = False
>>> print A1
>>> A2 = type(A2)
>>> print A1

(7) What will Python print?

>>> wall,table,lamp,desk = ("plane","K",[True,2,0],"neighbor")
>>> print len(table)*len(lamp)*len(desk)

(8) What will be printed?

>>> Camp = [True,True,77,88]
>>> Camp += 99
>>> print Camp
(9) What is the printed output for the following?

```python
>>> t3,t4 = "going going gone", ["found","here"]
>>> t3 = t3.split()
>>> t3 += t4
>>> print t3
```

(10) What is printed?

```python
>>> a = b = [4,3,5,2,1,6,7]
>>> a[-1:] = [False,False]
>>> print b
```

(11) For this question, first a function definition is given, then the interactive part follows.

```python
def YR(z):
    if len(z[0])>0:
        z[0] = z[0].upper()

>>> X = Y = "easy beat for dancing".split()
>>> YR(Y)
>>> print " ".join(X+Y)
```

(12) What’s printed here?

```python
>>> A2 = {"A":9, "B":8, "C":7, "D":6}
>>> A1 = A2
>>> A2[0] = True
>>> print A1
>>> A2 = type(A2)
>>> print A1
```

(13) What gets printed?

```python
>>> E = [1,2,3,4]
>>> E[3],E[2],E[1],E[0] = ("top","bottom",[5,6,7],"side")
>>> print len(E[2])*min(len(E[0]),len(E[1]))
```

(14) What is printed by this interactive session?

```python
>>> def = "radical definition".split()
>>> def += ["more"]
>>> print def
```

(15) What is printed by the Python session below?
>>> X, Y = ([3, 2, 5], [0, 0, 1])
>>> Y[0:0] = [-1]
>>> X += Y
>>> print X

(16) What will be printed?

>>> a = b = "tag along with me".split()
>>> a[0] = "retag"
>>> print b

(17) For this question, first a function definition is given, then the interactive part follows.

```python
def YR(z):
    z[0] += z[0]
```

>>> X = Y = "easy beat for dancing".split()
>>> YR(Y)
>>> print " ".join(X+Y)

(18) What will be printed?

>>> A2 = {"A":9, "B":8, "C":7, "D":6}
>>> A1 = A2
>>> del A2["B"]
>>> A1["B"] = 5
>>> print A1
>>> A2 = 
>>> print A1

(19) What is printed?

```python
>>> a, b, c, d = True, False, 1, 0
>>> E = [a, b, c, d]
>>> a, b = 7, 8
>>> print E
```

(20) What will Python print?

```python
>>> show, print = 0, 1
>>> print += 3
>>> print show, print
```

(21) What does this print?
>>> W, WW, WWW = ("three", "two", "one")
>>> WW[0] = "X"
>>> print W + WW + WWW

(22) What gets printed here?

>>> a = b = "this can be played two ways".split()
>>> a[0], a[-1] = "that", "times"
>>> print b

(23) For this question, first a function definition is given, then the interactive part follows.

def YR(z):
    z[0], z[1] = z[1], z[0]

>>> X = Y = "easy beat for dancing".split()
>>> YR(Y)
>>> print " ".join(X)

(24) What will Python print?

>>> A2 = {"A":9, "B":8, "C":7, "D":6}
>>> A1 = A2
>>> del A1["C"]
>>> A1["B"] = 5
>>> print A1
>>> A2 = ""
>>> print A1