Module 7 - Overview

Introduction

Module Learning Outcomes

After successful completion of this module, you will be able to ...

1. Create and use tuples.
2. Explain the difference between mutable types and immutable types.
3. Use some techniques for mutating a list.
4. Explain the difference between rebinding a variable and mutating the value that a variable refers to.

Key questions:

* What does it mean when we say that Python variables are object references?
* What determines whether an object is mutable or immutable?

Explorations

Use the pages within this module to explore the following concepts:

* Exploration: [Tuples, mutability vs. immutability](https://canvas.oregonstate.edu/courses/1928696/pages/exploration-tuples-and-mutability-vs-immutability) (CLO 1c, MLOs 1-3)
* Exploration: [Object references and identity](https://canvas.oregonstate.edu/courses/1928696/pages/exploration-object-references-and-identity) (MLO 4)
* [Video demos](https://canvas.oregonstate.edu/courses/1928696/pages/module-7-video-demos) (CLO 1c, MLOs 1-4)
* [Module 7 Exercise Solutions](https://canvas.oregonstate.edu/courses/1928696/pages/module-7-exercise-solutions)

Optional Resources

* [*Think Python* Chapter 12Links to an external site.](http://greenteapress.com/thinkpython2/html/thinkpython2013.html)
* [*Think Python* Chapter 10, sections 10 - 11Links to an external site.](http://greenteapress.com/thinkpython2/html/thinkpython2011.html#sec123)

Task List

Complete the following assignments and other tasks:

* Read the Exploration pages and do the interactive exercises on those pages (CLO 1c, MLOs 1-4).
* Assignment 7 gives you practice with mutating lists and the difference between mutating a value and rebinding a variable (CLO 1c, MLOs 1-4).
* Do [Assignment 7.a, b](https://canvas.oregonstate.edu/courses/1928696/assignments/9073290) (CLO 1c, MLOs 1-3)
* Then do [Assignment 7.c](https://canvas.oregonstate.edu/courses/1928696/assignments/9073291) (MLO 4)
* Take [Quiz 7](https://canvas.oregonstate.edu/courses/1928696/quizzes/2825534)  (CLO 1c, MLOs 1-4).

# Exploration: Tuples and Mutability vs. Immutability

## Tuples

Tuples are almost the same thing as a list. There are two minor differences having to do with syntax, and one more significant difference. Let's talk about the syntax first. The first difference is that we use square brackets to create a new list, but we use parentheses to create a new tuple.

The other syntax difference is with tuples that contain only one value. In order for Python to know that it's a tuple, you have to include a trailing comma.

Although parentheses are used for creating a new tuple, we still use square brackets when indexing or slicing a tuple.

Now let's look at the big difference.

## Mutability vs. immutability

The big important difference between lists and tuples is that lists are mutable, but tuples are immutable. Put more simply, you can change a list, but you can't change a tuple. So far we've seen ways to create new lists based on old lists, but we haven't looked at ways we can change an existing list, with one exception. The sort method doesn't create a new sorted version of the list - it rearranges the existing list into sorted order.

> numbers.sort()

We can't call sort() on  tuples because they're immutable.

### **Mutating a list**

How else can we mutate (modify) an existing list? Here are a couple of lists we can experiment with.

We can assign a new value to an existing element of a list like this:

> members  
> members[0] = "Marky" # using indexing to mutate a list  
> members

We can also put a slice on the left side of the assignment. In this case the value on the right must be of an iterable type.

> birds[1:3] = ["robin", "chickadee"] # using slicing to mutate a list  
> birds

The number of elements being assigned can be different than the size of the slice being replaced. (Enter "birds" after each example so that you can see what has changed.)

> birds[1:3] = ["hummingbird", "wren", "emu", "penguin"]  
  
> birds[3:6] = ["cassowary"]  
  
> birds[1:1] = ["kiwi", "big bird"]  
  
> birds[2:3] = []

Notice that the slice "birds[1:1]" is empty, since a slice goes up to the second index, but doesn't include it.

The slice "birds[:]" would create a slice of the entire list - that is, it would create a copy of the list, which can also be done using list():

> birds\_copy\_1 = birds[:]  
> birds\_copy\_2 = list(birds)

We can also append items to the end of a list using **append**, and delete items from a list using **del**(notice that append uses method notation, but del uses operator notation).

> vocab\_words = []  
> vocab\_words.append("usagi")  
> vocab\_words.append("inazuma")  
> vocab\_words.append("hebi")  
> vocab\_words.append("kitsune")  
> vocab\_words  
> del vocab\_words[2]  
> vocab\_words

### **What about tuples?**

None of these assignments will work with tuples because they cannot be mutated. But if tuples are essentially lists that we can't change, why use them at all? One reason is that there are times when an immutable type is required. For example, only immutable types can be used as keys in **dictionaries**, another way of storing and manipulating data that we'll talk about soon. Another reason is that if the elements shouldn't get changed, using a tuple instead of a list makes sure that won't happen by accident. It can also communicate to someone reading or updating your code that the collection is not intended to be modified.

### **Immutable collections of mutable objects**

If you have a tuple that contains mutable objects, the tuple itself is immutable because you cannot change which objects it contains, however the mutable objects it contains can still be mutated.

### **Mutability and immutability of function arguments**

In the page on functions we saw that changing a parameter in a function did not change the value of the variable that was passed in the function call, and said we'd discuss the reason in a later module. The reason is because the variable we passed referred to an immutable type. Integers, like tuples, are immutable and cannot be changed. In fact, lists are the **only** mutable type we've looked at so far - all the others we've seen are immutable (objects of user-defined classes, which we looked at in module 5, are also mutable). If a variable refers to a list, and we pass that list to a function, then the function can change the actual list that was passed.

Here we see that the variable num\_list got changed by the function we passed it to. The function could have changed the list in other ways such as adding or removing elements, but I wanted to show an example very similar to the one I showed in the page on functions, only using a list instead of an integer.

You might have an objection at this point. How can ints, floats, bools and strings be immutable? It seems like we've changed values of those types before. We'll take a look at this in the next section, which talks about object references and identity.

### **Mutable default arguments**

**Beware of this gotcha**: In Python, if you use a mutable value as a default argument, a new object won't be created every time the function is called.  Instead, a new object will be created the first time the function is called without an argument, and that **same** object will be used for later calls to the function (that don't have an argument).  So if you have a list as a default argument, the same list would be used every time the function is called without that argument, which is probably not what you would expect or want.  For example:

def some\_func(my\_list=[]):  
 my\_list.append(1)  
 return my\_list

The first time you call it without an argument, it will return [1].  The next time you call it without an argument, it will return [1, 1], then [1, 1, 1], etc.  In order to avoid this behavior, do the following instead:

def some\_func(my\_list=None):  
 if my\_list is None:  
 my\_list = []  
 my\_list.append(1)  
 return my\_list

Remember that when you use a default argument, the value on the right side of the '=' is the default **argument**, while the value on the left side is the **parameter** you're assigning it to. In both examples, my\_list is the **parameter**. In the first example, [] is the default **argument**. Since that's a list, and lists are mutable, that makes it a mutable default argument. In the second example, None is the default **argument**. Since None is not mutable, it is not a mutable default argument. Since integers, floats, Booleans, strings, and tuples are not mutable, you don't need to worry about using None instead. In this course, the only mutable types you're concerned with are lists, dictionaries, and sets.

**Do not use mutable default arguments for any of the assignments in this course.**

### **Tuple unpacking**

Python gives you a syntactical shortcut for assigning the values of a tuple to variables. It looks like this:

some\_tuple = (1,2,3) # creating a tuple  
num\_1,num\_2,num\_3 = some\_tuple # unpacking the tuple

The second line assigns the first, second, and third values of the tuple to num\_1, num\_2, and num\_3 respectively. You can even use tuple unpacking to swap the values of two variables without using a third variable as a temporary holding spot:

num\_1, num\_2 = num\_2, num\_1

**Even without parentheses, the right side of the assignment is still interpreted as a tuple.**

You can also unpack lists (the more general term is "sequence unpacking").

## Exercises

(See the module overview for a link to example solutions.)

1. Does the following program contradict the idea that tuples are immutable? Why or why not?

def tuple\_madness(tup):  
 return tup[1:]

2. Write a function named insert\_front that takes as parameters a list and a value to add at the front of the list. It should not return anything - it should mutate the original list. For example, if the arguments passed to the function are [9, -55, 37] and "bob", then after calling the function, the list should now be ["bob", 9, -55, 37].

Sample input: [True, False], "Unknown"  
Expected output: ["Unknown", True, False]

3. Write a function named delete\_last that takes as a parameter a list and removes the last element from that list. It should not return anything - it should mutate the original list. For example, if the list passed to the function is [7, "joe", "apple", 9.81, False], then after calling the function, the list should be [7, "joe", "apple", 9.81].

Sample input: ["Unknown", True, False]  
Expected output: ["Unknown", True]

# Exploration: Object References and Identity

## Introduction

Remember how variables refer to values, and how all values in Python are objects? It's possible for two or more variables to refer to the same object. We can tell whether this is the case by using the **is** operator.

> num\_1 = 1000  
> num\_2 = 1000  
> num\_1 == num\_2 # Test whether num\_1 and num\_2 are equal  
> num\_1 is num\_2 # Test whether num\_1 and num\_2 refer to the same object  
> num\_1 = num\_2 # Now num\_1 and num\_2 refer to the same object  
> num\_1 is num\_2  
> num\_2 = -89 # Make num\_2 refer to a different object  
> num\_1 is num\_2

The first line creates an object for the value 1000 and **binds** the variable num\_1 to it. The second line creates another object for the value 1000 and binds the variable num\_2 to it. Next we use the == operator to test whether num\_1 and num\_2 refer to values that are equal, which they do. Then we use the is operator to test whether num\_1 and num\_2 refer to (are bound to) the same object, which they aren't. Next we assign num\_2 to num\_1. This **rebinds** num\_1, so that now it refers to the same object that num\_2 does. When we test again whether num\_1 and num\_2 refer to the same object, we see that they now do. Next we have another assignment statement, which creates an object for the value -89 and rebinds num\_2 to refer to that object. Now we see that num\_1 and num\_2 no longer refer to the same object. Assigning -89 to num\_2 **did not change** the object for the value 1000, which num\_1 still refers to. We cannot change the value of an integer object, because integers are immutable, but we can rebind a variable to a different object.

You can think of a variable as like a sticky note that you can stick to different objects. You can have multiple sticky notes attached (bound) to the same object, and you can take a sticky note off one object and attach it to another object, even an object of another type.

### **Example Using Lists**

Now let's look at an example that uses lists, which we know are mutable.

> num\_list\_1 == num\_list\_2 # Test whether num\_list\_1 and num\_list\_2 are equal  
> num\_list\_1 is num\_list\_2 # Test whether they refer to the same object  
> num\_list\_1 = num\_list\_2 # Now they refer to the same object  
> num\_list\_1 is num\_list\_2  
> num\_list\_2 = [23,-1,19] # Make num\_list\_2 refer to a different object  
> num\_list\_1 is num\_list\_2

This example behaves the same way as the previous example. So far, being mutable doesn't seem to make a difference. However, let's try something a little different:

> num\_list\_1 = [3,17,12]  
> num\_list\_2 = num\_list\_1 # num\_list\_1 and num\_list\_2 refer to the same object  
> num\_list\_1 is num\_list\_2  
> num\_list\_2[0] = 99 # Change the object that num\_list\_1 and num\_list\_2 both refer to  
> num\_list\_1  
> num\_list\_1 is num\_list\_2

In this example, changing num\_list\_2 does **not** result in creating a new object and rebinding num\_list\_2 to that object. Instead, it mutates the existing list object, which num\_list\_1 and num\_list\_2 both still refer to. We can see that **there is a difference between rebinding a variable, and mutating the value that a variable refers to**.

### **Changing Parameters - Mutating Values**

This helps explain what happens when a function "changes" one of its parameters. Let's revisit a couple of examples we've seen previously:

In the first example, we pass num to the function and val gets bound to the same object as num. Then the line "val = val \* val" creates a new object for the value 64, and **rebinds val to that object**. At this point num and val no longer refer to the same object. This is why num doesn't show any change after the function call.

In the second example, we pass num\_list to the function and val gets bound to the same object as num\_list. Then the line "val[0] = val[0] \* val[0]" **mutates** the list that both num\_list and val refer to. This is why num\_list shows a change after the function call.

So if you need for a function to mutate a value, then that value needs to be of a mutable type.  Note that in Python a variable doesn't have a type - it's the value it references that has a type, and that type determines whether the value is mutable or immutable.

### **Shallow copies vs deep copies**

We can make a copy of a list like this:

> original\_list = [1, [1,2,3]]  
> copied\_list = list(original\_list)

copied\_list now holds a copy of original\_list, but it's just a shallow copy.

> original\_list is copied\_list # False  
> original\_list[0] is copied\_list[0] # True  
> original\_list[1] is copied\_list[1] # True

We can see that although original\_list and copied\_list refer to different objects, the elements of original\_list and the elements of copied\_list refer to the same objects.  If you mutate the list referenced by original\_list[1], and then print copied\_list:

> original\_list[1][0] = 99  
> copied\_list

You'll see that copied\_list shows that change as well.

A deep copy, however, would be an entirely separate copy (no shared sub-elements or sub-sub-elements, etc.).  If we had made a deep copy in this example instead of a shallow copy, it would look like this:

We could have achieved a deep copy by also making a copy of any sublists (or other mutable sub-elements):

copied\_list = list(original\_list)  
copied\_list[1] = list(original\_list[1])

There is a Python module that can be used to make deep copies, but we won't cover that here.

Continuing with our shallow copy example, if you enter the following:

> original\_list[0] = 5  
> copied\_list

You'll see that copied\_list hasn't changed.  That's because we're not mutating the object referred to by original\_list[0] (it's an int, and ints are immutable).  Instead, we're rebinding original\_list[0] to a new int object.

We were able to mutate the object referred to by original\_list[1] because it's a list, and lists are mutable.  If we wanted, we could rebind original\_list[1], like this:

> original\_list[1] = [2,4,6,8]

In which case copied\_list wouldn't be affected, because we're not mutating a shared list, but instead rebinding original\_list[1] to a new list object.

In practice, you'll nearly always want a deep copy rather than a shallow copy, though if you have an immutable data structure, such as a tuple, and all of the data types in it are also immutable, then a shallow copy may suffice for your purposes.

### **Automatic sharing**

Sometimes Python is able to figure out that it can save memory by automatically making variables refer to the same object.

> num\_1 = 5  
> num\_2 = 5  
> num\_1 is num\_2  
  
> name\_1 = "Betty"  
> name\_2 = "Betty"  
> name\_1 is name\_2

This only ever happens with (equal) immutable values, so it can't affect your code's results.

You might notice that this example using 5 gives you different results than the earlier example that used 1000.  It depends too on whether you're running the code in the interactive console or running it as a script.  In the interactive console, Python keeps an array of integer objects for -5 through 256.  When you create an int in that range, you actually just get back a reference to the existing object.  So if you do that with two (or more) variables, they will then refer to the same object.  However, if you type that code as a script in PyCharm and then try running it with different values, you can see that any integer values will be shared, not just from -5 to 256.  This is because when you're running a script, the Python interpreter can look through the entire script before running it, so it knows all the values that could be shared.

## Exercises

(See the module overview for a link to example solutions.)

1. Is it possible to have variables be equal, but not refer to the same object? Is it possible to have variables refer to the same object, but not be equal? Explain both answers.
2. If nums\_1 and nums\_2 refer to the same **list** object, and you delete the first element of nums\_1, then that affects nums\_2, because they're the same list. Why is this not true if nums\_1 and nums\_2 refer to the same **tuple** object?
3. When does rebinding of a variable happen?