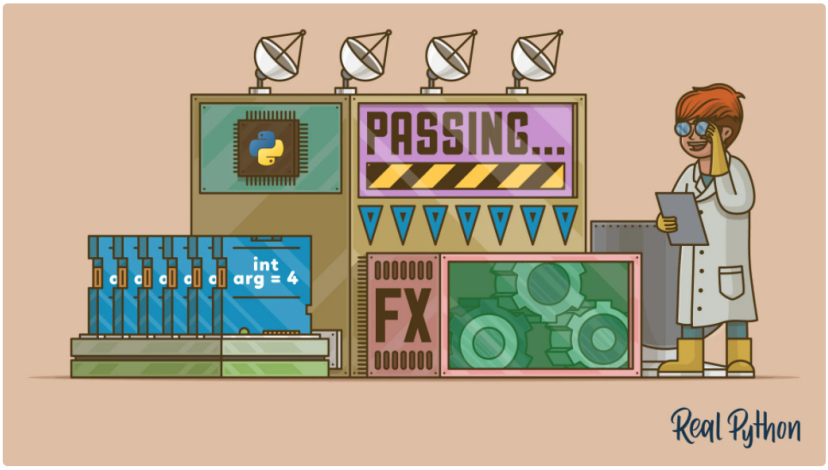

Pass by Reference in Python: Background and Best Practices

by Marius Mogyorosi Aug 10, 2020 9 Comments best-practices intermediate python

Mark as Completed

Tweet Share Email

Table of Contents

- Defining Pass by Reference
- Contrasting Pass by Reference and Pass by Value
- Using Pass by Reference Constructs
  - Avoiding Duplicate Objects
  - Returning Multiple Values
  - Creating Conditional Multiple-Return Functions
- Passing Arguments in Python
  - Understanding Assignment in Python
  - Exploring Function Arguments
- Replicating Pass by Reference With Python
  - Best Practice: Return and Reassign
  - Best Practice: Use Object Attributes
  - Best Practice: Use Dictionaries and Lists
- Conclusion

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Pass by Reference in Python: Best Practices**

After gaining some familiarity with Python, you may notice cases in which your functions don't modify arguments in place as you might expect, especially if you're familiar with other programming languages. Some languages handle function arguments as **references** to existing variables, which is known as **pass by reference**. Other languages handle them as **independent values**, an approach known as **pass by value**.

If you're an intermediate Python programmer who wishes to understand Python's peculiar way of handling function arguments, then this tutorial is for you. You'll implement real use cases of pass-by-reference constructs in Python and learn several best practices to avoid pitfalls with your function arguments.

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents

- Defining Pass by Reference
- Contrasting Pass by Reference and Pass by Value
- Using Pass by Reference Constructs
- Passing Arguments in Python
- Replicating Pass by Reference With Python
- Conclusion

Mark as Completed

Tweet Share Email

Recommended Video Course
Pass by Reference in Python: Best Practices

In this tutorial, you'll learn:

- What it means to **pass by reference** and why you'd want to do so
- How passing by reference differs from both **passing by value** and **Python's unique approach**
- How **function arguments** behave in Python
- How you can use certain **mutable types** to pass by reference in Python
- What the **best practices** are for replicating pass by reference in Python

Defining Pass by Reference

Before you dive into the technical details of passing by reference, it's helpful to take a closer look at the term itself by breaking it down into components:

- **Pass** means to provide an argument to a function.
- **By reference** means that the argument you're passing to the function is a **reference** to a variable that already exists in memory rather than an independent copy of that variable.

Since you're giving the function a reference to an existing variable, all operations performed on this reference will directly affect the variable to which it refers. Let's look at some examples of how this works in practice.

Below, you'll see how to pass variables by reference in C#. Note the use of the `ref` keyword in the highlighted lines:

C#

```
using System;

// Source:
// https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-struct
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

As you can see, the `refParameter` of `squareRef()` must be declared with the `ref` keyword, and you must also use the keyword when calling the function. Then the argument will be passed in by reference and can be modified in place.

Python has no `ref` keyword or anything equivalent to it. If you attempt to replicate the above example as closely as possible in Python, then you'll see different results:

Python >>>

```
>>> def main():
...     arg = 4
...     square(arg)
```

```
...     print(arg)
...
>>> def square(n):
...     n *= n
...
>>> main()
4
```

In this case, the `arg` variable is *not* altered in place. It seems that Python treats your supplied argument as a standalone value rather than a reference to an existing variable. Does this mean Python passes arguments by value rather than by reference?

Not quite. Python passes arguments neither by reference nor by value, but **by assignment**. Below, you'll quickly explore the details of passing by value and passing by reference before looking more closely at Python's approach. After that, you'll walk through some best practices for achieving the equivalent of passing by reference in Python.

Contrasting Pass by Reference and Pass by Value

When you pass function arguments by reference, those arguments are only references to existing values. In contrast, when you pass arguments by value, those arguments become independent copies of the original values.

Let's revisit the C# example, this time without using the `ref` keyword. This will cause the program to use the default behavior of passing by value:

C#

```
using System;

// Source:
// https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-struct
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }
}
```

Here, you can see that `squareVal()` doesn't modify the original variable. Rather, `valParameter` is an independent copy of the original variable `arg`. While that matches the behavior you would see in Python, remember that Python doesn't exactly pass by value. Let's prove it.

Python's built-in `id()` returns an integer representing the memory address of the desired object. Using `id()`, you can verify the following assertions:

1. Function arguments initially refer to the same address as their original variables.
2. Reassigning the argument within the function gives it a new address while the original variable remains unmodified.

In the below example, note that the address of `x` initially matches that of `n` but changes after

reassignment, while the address of `n` never changes:

```
Python >>>
>>> def main():
...     n = 9001
...     print(f"Initial address of n: {id(n)}")
...     increment(n)
...     print(f"  Final address of n: {id(n)}")
...
>>> def increment(x):
...     print(f"Initial address of x: {id(x)}")
...     x += 1
...     print(f"  Final address of x: {id(x)}")
...
>>> main()
Initial address of n: 140562586057840
Initial address of x: 140562586057840
  Final address of x: 140562586057968
  Final address of n: 140562586057840
```

The fact that the initial addresses of `n` and `x` are the same when you invoke `increment()` proves that the `x` argument is not being passed by value. Otherwise, `n` and `x` would have distinct memory addresses.

Before you learn the details of how Python handles arguments, let's take a look at some practical use cases of passing by reference.

Using Pass by Reference Constructs

Passing variables by reference is one of several strategies you can use to implement certain programming patterns. While it's seldom necessary, passing by reference can be a useful tool.

In this section, you'll look at three of the most common patterns for which passing by reference is a practical approach. You'll then see how you can implement each of these patterns with Python.

Avoiding Duplicate Objects

As you've seen, passing a variable by value will cause a copy of that value to be created and stored in memory. In languages that default to passing by value, you may find performance benefits from passing the variable by reference instead, especially when the variable holds a lot of data. This will be more apparent when your code is running on resource-constrained machines.

In Python, however, this is never a problem. You'll see why in the next section.

Returning Multiple Values

One of the most common applications of passing by reference is to create a function that alters the value of the reference parameters while returning a distinct value. You can modify your pass-by-reference C# example to illustrate this technique:

```
C#
using System;

class Program
{
    static void Main(string[] args)
    {
        int counter = 0;

        // Passing by reference.
        // The value of counter in Main is changed.
        Console.WriteLine(greet("Alice", ref counter));
        Console.WriteLine("Counter is {0}", counter);
        Console.WriteLine(greet("Bob", ref counter));
        Console.WriteLine("Counter is {0}", counter);
        // Output:
        // Hi, Alice!
```

```
        // Counter is 1
        // Hi, Bob!
        // Counter is 2
    }

    static string greet(string name, ref int counter)
    {
        string greeting = "Hi, " + name + "!";
        counter++;
        return greeting;
    }
}
```

In the example above, `greet()` returns a greeting string and also modifies the value of `counter`. Now try to reproduce this as closely as possible in Python:

Python >>>

```
>>> def main():
...     counter = 0
...     print(greet("Alice", counter))
...     print(f"Counter is {counter}")
...     print(greet("Bob", counter))
...     print(f"Counter is {counter}")
...
>>> def greet(name, counter):
...     counter += 1
...     return f"Hi, {name}!"
...
>>> main()
Hi, Alice!
Counter is 0
Hi, Bob!
Counter is 0
```

`counter` isn't incremented in the above example because, as you've previously learned, Python has no way of passing values by reference. So how can you achieve the same outcome as you did with C#?

In essence, reference parameters in C# allow the function not only to return a value but also to operate on additional parameters. This is equivalent to returning multiple values!

Luckily, Python already supports returning multiple values. Strictly speaking, a Python function that returns multiple values actually returns a tuple containing each value:

Python >>>

```
>>> def multiple_return():
...     return 1, 2
...
>>> t = multiple_return()
>>> t  # A tuple
(1, 2)

>>> # You can unpack the tuple into two variables:
>>> x, y = multiple_return()
>>> x
1
>>> y
2
```

As you can see, to return multiple values, you can simply use the `return` keyword followed by comma-separated values or variables.

Armed with this technique, you can change the `return` statement in `greet()` from your previous Python code to return both a greeting and a counter:

Python >>>

```
>>> def main():
...     counter = 0
...     print(greet("Alice", counter))
...     print(f"Counter is {counter}")
...     print(greet("Bob", counter))
...     print(f"Counter is {counter}")
```

```
...     print(f"Counter is {counter}")
...
>>> def greet(name, counter):
...     return f"Hi, {name}!", counter + 1
...
>>> main()
('Hi, Alice!', 1)
Counter is 0
('Hi, Bob!', 1)
Counter is 0
```

That still doesn't look right. Although `greet()` now returns multiple values, they're being printed as a `tuple`, which isn't your intention. Furthermore, the original `counter` variable remains at `0`.

To clean up your output and get the desired results, you'll have to **reassign** your `counter` variable with each call to `greet()`:

Python >>>

```
>>> def main():
...     counter = 0
...     greeting, counter = greet("Alice", counter)
...     print(f"{greeting}\nCounter is {counter}")
...     greeting, counter = greet("Bob", counter)
...     print(f"{greeting}\nCounter is {counter}")
...
>>> def greet(name, counter):
...     return f"Hi, {name}!", counter + 1
...
>>> main()
Hi, Alice!
Counter is 1
Hi, Bob!
Counter is 2
```

Now, after reassigning each variable with a call to `greet()`, you can see the desired results!

Assigning return values to variables is the best way to achieve the same results as passing by reference in Python. You'll learn why, along with some additional methods, in the section on best practices.

Creating Conditional Multiple-Return Functions

This is a specific use case of returning multiple values in which the function can be used in a conditional statement and has additional side effects like modifying an external variable that was passed in as an argument.

Consider the standard Int32.TryParse function in C#, which returns a Boolean and operates on a reference to an integer argument at the same time:

C#

```
public static bool TryParse (string s, out int result);
```

This function attempts to convert a `string` into a 32-bit signed integer using the `out` keyword. There are two possible outcomes:

1. **If parsing succeeds**, then the output parameter will be set to the resulting integer, and the function will return `true`.
2. **If parsing fails**, then the output parameter will be set to `0`, and the function will return `false`.

You can see this in practice in the following example, which attempts to convert a number of different strings:

C#

```
using System;

// Source:
// https://docs.microsoft.com/en-us/dotnet/api/system.int32.tryparse?view=netcore-3.1
public class Example {
    public static void Main() {
        String[] values = { null, "160519", "9432.0", "16,667",
                            "   -322   ", "+4302", "(100);", "01FA" };
        foreach (var value in values) {
            int number;

            if (Int32.TryParse(value, out number)) {
                Console.WriteLine("Converted '{0}' to {1}.", value, number);
            }
            else {
                Console.WriteLine("Attempted conversion of '{0}' failed.",
                                  value ?? "<null>");
            }
        }
    }
}
```

The above code, which attempts to convert differently formatted strings into integers via `TryParse()`, outputs the following:

Shell

```
Attempted conversion of '<null>' failed.
Converted '160519' to 160519.
Attempted conversion of '9432.0' failed.
Attempted conversion of '16,667' failed.
Converted '   -322   ' to -322.
Converted '+4302' to 4302.
Attempted conversion of '(100);' failed.
Attempted conversion of '01FA' failed.
```

To implement a similar function in Python, you could use multiple return values as you've seen previously:

Python

```
def tryparse(string, base=10):
    try:
        return True, int(string, base=base)
    except ValueError:
        return False, None
```

This `tryparse()` returns two values. The first value indicates whether the conversion was successful, and the second holds the result (or `None`, in case of failure).

However, using this function is a little clunky because you need to unpack the return values with every call. This means you can't use the function within an `if` statement:

Python >>>

```
>>> success, result = tryparse("123")
>>> success
True
>>> result
123

>>> # We can make the check work
>>> # by accessing the first element of the returned tuple,
>>> # but there's no way to reassign the second element to `result`:
>>> if tryparse("456")[0]:
...     print(result)
...
123
```

Even though it generally works by returning multiple values, `tryparse()` can't be used in a condition check. That means you have some more work to do.

You can take advantage of Python's flexibility and simplify the function to return a single

value of different types depending on whether the conversion succeeds:

```python
def tryparse(string, base=10):
    try:
        return int(string, base=base)
    except ValueError:
        return None
```

With the ability for Python functions to return different data types, you can now use this function within a conditional statement. But how? Wouldn't you have to call the function first, assigning its return value, and then check the value itself?

By taking advantage of Python's flexibility in object types, as well as the new assignment expressions in Python 3.8, you can call this simplified function within a conditional `if` statement *and* get the return value if the check passes:

```
>>> if (n := tryparse("123")) is not None:
...     print(n)
...
123
>>> if (n := tryparse("abc")) is not None:
...     print(n)
...
None

>>> # You can even do arithmetic!
>>> 10 * tryparse("10")
100

>>> # All the functionality of int() is available:
>>> 10 * tryparse("0a", base=16)
100

>>> # You can also embed the check within the arithmetic expression!
>>> 10 * (n if (n := tryparse("123")) is not None else 1)
1230
>>> 10 * (n if (n := tryparse("abc")) is not None else 1)
10
```

Wow! This Python version of `tryparse()` is even more powerful than the C# version, allowing you to use it within conditional statements and in arithmetic expressions.

With a little ingenuity, you've replicated a specific and useful pass-by-reference pattern without actually passing arguments by reference. In fact, you are yet again **assigning return values** when using the assignment expression operator(`:=`) and using the return value directly in Python expressions.

So far, you've learned what passing by reference means, how it differs from passing by value, and how Python's approach is different from both. Now you're ready to take a closer look at how Python handles function arguments!

Passing Arguments in Python

Python passes arguments by assignment. That is, when you call a Python function, each function argument becomes a variable to which the passed value is assigned.

Therefore, you can learn important details about how Python handles function arguments by understanding how the assignment mechanism itself works, even outside functions.

Understanding Assignment in Python

Python's language reference for assignment statements provides the following details:

- If the assignment target is an identifier, or variable name, then this name is bound to the object. For example, in `x = 2`, `x` is the name and `2` is the object.
- If the name is already bound to a separate object, then it's re-bound to the new object. For example, if `x` is already `2` and you issue `x = 3`, then the variable name `x` is re-bound to `3`.

All Python objects are implemented in a particular structure. One of the properties of this structure is a counter that keeps track of how many names have been bound to this object.

> **Note:** This counter is called a **reference counter** because it keeps track of how many references, or names, point to the same object. Do not confuse reference counter with the concept of passing by reference, as the two are unrelated.
>
> The Python documentation provides additional details on reference counts.

Let's stick to the `x = 2` example and examine what happens when you assign a value to a new variable:

1. If an object representing the value `2` already exists, then it's retrieved. Otherwise, it's created.
2. The reference counter of this object is incremented.
3. An entry is added in the current namespace to bind the identifier `x` to the object representing `2`. This entry is in fact a key-value pair stored in a dictionary! A representation of that dictionary is returned by `locals()` or `globals()`.

Now here's what happens if you reassign `x` to a different value:

1. The reference counter of the object representing `2` is decremented.
2. The reference counter of the object that represents the new value is incremented.
3. The dictionary for the current namespace is updated to relate `x` to the object representing the new value.

Python allows you to obtain the reference counts for arbitrary values with the function `sys.getrefcount()`. You can use it to illustrate how assignment increases and decreases these reference counters. Note that the interactive interpreter employs behavior that will yield different results, so you should run the following code from a file:

Python

```python
from sys import getrefcount

print("--- Before  assignment ---")
print(f"References to value_1: {getrefcount('value_1')}")
print(f"References to value_2: {getrefcount('value_2')}")
x = "value_1"
print("--- After   assignment ---")
print(f"References to value_1: {getrefcount('value_1')}")
print(f"References to value_2: {getrefcount('value_2')}")
x = "value_2"
print("--- After reassignment ---")
print(f"References to value_1: {getrefcount('value_1')}")
print(f"References to value_2: {getrefcount('value_2')}")
```

This script will show the reference counts for each value prior to assignment, after assignment, and after reassignment:

Shell

```
--- Before  assignment ---
References to value_1: 3
References to value_2: 3
--- After   assignment ---
References to value_1: 4
References to value_2: 3
--- After reassignment ---
References to value_1: 3
References to value_2: 4
```

These results illustrate the relationship between identifiers (variable names) and Python objects that represent distinct values. When you assign multiple variables to the same value, Python increments the reference counter for the existing object and updates the current namespace rather than creating duplicate objects in memory.

In the next section, you'll build upon your current understanding of assignment operations by exploring how Python handles function arguments.

Exploring Function Arguments

Function arguments in Python are **local variables**. What does that mean? **Local** is one of Python's scopes. These scopes are represented by the namespace dictionaries mentioned in the previous section. You can use `locals()` and `globals()` to retrieve the local and global namespace dictionaries, respectively.

Upon execution, each function has its own local namespace:

Python >>>

```
>>> def show_locals():
...     my_local = True
...     print(locals())
...
>>> show_locals()
{'my_local': True}
```

Using `locals()`, you can demonstrate that function arguments become regular variables in the function's local namespace. Let's add an argument, `my_arg`, to the function:

Python >>>

```
>>> def show_locals(my_arg):
...     my_local = True
...     print(locals())
...
>>> show_locals("arg_value")
{'my_arg': 'arg_value', 'my_local': True}
```

You can also use `sys.getrefcount()` to show how function arguments increment the reference counter for an object:

Python >>>

```
>>> from sys import getrefcount

>>> def show_refcount(my_arg):
...     return getrefcount(my_arg)
...
>>> getrefcount("my_value")
3
>>> show_refcount("my_value")
5
```

The above script outputs reference counts for `"my_value"` first outside, then inside `show_refcount()`, showing a reference count increase of not one, but two!

That's because, in addition to `show_refcount()` itself, the call to `sys.getrefcount()` inside `show_refcount()` also receives `my_arg` as an argument. This places `my_arg` in the local namespace for `sys.getrefcount()`, adding an extra reference to `"my_value"`.

By examining namespaces and reference counts inside functions, you can see that function arguments work exactly like assignments: Python creates bindings in the function's local namespace between identifiers and Python objects that represent argument values. Each of these bindings increments the object's reference counter.

Now you can see how Python passes arguments by assignment!

Replicating Pass by Reference With Python

Having examined namespaces in the previous section, you may be asking why `global` hasn't been mentioned as one way to modify variables as if they were passed by reference:

Python >>>

```
>>> def square():
...     # Not recommended!
...     global n
...     n *= n
...
>>> n = 4
>>> square()
>>> n
16
```

Using the `global` statement generally takes away from the clarity of your code. It can create a number of issues, including the following:

- Free variables, seemingly unrelated to anything
- Functions without explicit arguments for said variables
- Functions that can't be used generically with other variables or arguments since they rely on a single global variable
- Lack of thread safety when using global variables

Contrast the previous example with the following, which explicitly returns a value:

Python >>>

```
>>> def square(n):
...     return n * n
...
>>> square(4)
16
```

Much better! You avoid all potential issues with global variables, and by requiring an argument, you make your function clearer.

Despite being neither a pass-by-reference language nor a pass-by-value language, Python suffers no shortcomings in that regard. Its flexibility more than meets the challenge.

Best Practice: Return and Reassign

You've already touched on returning values from the function and reassigning them to a variable. For functions that operate on a single value, returning the value is much clearer than using a reference. Furthermore, since Python already uses pointers behind the scenes, there would be no additional performance benefits even if it were able to pass arguments by reference.

Aim to write single-purpose functions that return one value, then (re)assign that value to variables, as in the following example:

Python

```
def square(n):
    # Accept an argument, return a value.
    return n * n

x = 4
...
# Later, reassign the return value:
x = square(x)
```

Returning and assigning values also makes your intention explicit and your code easier to understand and test.

For functions that operate on multiple values, you've already seen that Python is capable of

returning a tuple of values. You even surpassed the elegance of Int32.TryParse() in C# thanks to Python's flexibility!

If you need to operate on multiple values, then you can write single-purpose functions that return multiple values, then (re)assign those values to variables. Here's an example:

Python

```
def greet(name, counter):
    # Return multiple values
    return f"Hi, {name}!", counter + 1

counter = 0
...
# Later, reassign each return value by unpacking.
greeting, counter = greet("Alice", counter)
```

When calling a function that returns multiple values, you can assign multiple variables at the same time.

Best Practice: Use Object Attributes

Object attributes have their own place in Python's assignment strategy. Python's language reference for assignment statements states that if the target is an object's attribute that supports assignment, then the object will be asked to perform the assignment on that attribute. If you pass the object as an argument to a function, then its attributes can be modified in place.

Write functions that accept objects with attributes, then operate directly on those attributes, as in the following example:

Python >>>

```
>>> # For the purpose of this example, let's use SimpleNamespace.
>>> from types import SimpleNamespace

>>> # SimpleNamespace allows us to set arbitrary attributes.
>>> # It is an explicit, handy replacement for "class X: pass".
>>> ns = SimpleNamespace()

>>> # Define a function to operate on an object's attribute.
>>> def square(instance):
...     instance.n *= instance.n
...
>>> ns.n = 4
>>> square(ns)
>>> ns.n
16
```

Note that `square()` needs to be written to operate directly on an attribute, which will be modified without the need to reassign a return value.

It's worth repeating that you should make sure the attribute supports assignment! Here's the same example with `namedtuple`, whose attributes are read-only:

Python >>>

```
>>> from collections import namedtuple
>>> NS = namedtuple("NS", "n")
>>> def square(instance):
...     instance.n *= instance.n
...
>>> ns = NS(4)
>>> ns.n
4
>>> square(ns)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in square
AttributeError: can't set attribute
```

Attempts to modify attributes that don't allow modification result in an `AttributeError`.

Additionally, you should be mindful of class attributes. They will remain unchanged, and an instance attribute will be created and modified:

```
Python >>>
>>> class NS:
...     n = 4
...
>>> ns = NS()
>>> def square(instance):
...     instance.n *= instance.n
...
>>> ns.n
4
>>> square(ns)
>>> # Instance attribute is modified.
>>> ns.n
16
>>> # Class attribute remains unchanged.
>>> NS.n
4
```

Since class attributes remain unchanged when modified through a class instance, you'll need to remember to reference the instance attribute.

Best Practice: Use Dictionaries and Lists

Dictionaries in Python are a different object type than all other built-in types. They're referred to as **mapping types**. Python's documentation on mapping types provides some insight into the term:

> A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. (Source)

This tutorial doesn't cover how to implement a custom mapping type, but you can replicate pass by reference using the humble dictionary. Here's an example using a function that operates directly on dictionary elements:

```
Python >>>
>>> # Dictionaries are mapping types.
>>> mt = {"n": 4}
>>> # Define a function to operate on a key:
>>> def square(num_dict):
...     num_dict["n"] *= num_dict["n"]
...
>>> square(mt)
>>> mt
{'n': 16}
```

Since you're reassigning a value to a dictionary key, operating on dictionary elements is still a form of assignment. With dictionaries, you get the added practicality of accessing the modified value through the same dictionary object.

While lists aren't mapping types, you can use them in a similar way to dictionaries because of two important characteristics: **subscriptability** and **mutability**. These characteristics are worthy of a little more explanation, but let's first take a look at best practices for mimicking pass by reference using Python lists.

To replicate pass by reference using lists, write a function that operates directly on list elements:

```
Python >>>
>>> # Lists are both subscriptable and mutable.
>>> sm = [4]
```

```
>>> # Define a function to operate on an index:
>>> def square(num_list):
...     num_list[0] *= num_list[0]
...
>>> square(sm)
>>> sm
[16]
```

Since you're reassigning a value to an element within the list, operating on list elements is still a form of assignment. Similar to dictionaries, lists allow you to access the modified value through the same list object.

Now let's explore subscriptability. An object is **subscriptable** when a subset of its structure can be accessed by index positions:

Python >>>

```
>>> subscriptable = [0, 1, 2]  # A list
>>> subscriptable[0]
0
>>> subscriptable = (0, 1, 2)  # A tuple
>>> subscriptable[0]
0
>>> subscriptable = "012"  # A string
>>> subscriptable[0]
'0'
>>> not_subscriptable = {0, 1, 2}  # A set
>>> not_subscriptable[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

Lists, tuples, and strings are subscriptable, but sets are not. Attempting to access an element of an object that isn't subscriptable will raise a `TypeError`.

Mutability is a broader topic requiring additional exploration and documentation reference. To keep things short, an object is **mutable** if its structure can be changed in place rather than requiring reassignment:

Python >>>

```
>>> mutable = [0, 1, 2]  # A list
>>> mutable[0] = "x"
>>> mutable
['x', 1, 2]

>>> not_mutable = (0, 1, 2)  # A tuple
>>> not_mutable[0] = "x"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> not_mutable = "012"  # A string
>>> not_mutable[0] = "x"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> mutable = {0, 1, 2}  # A set
>>> mutable.remove(0)
>>> mutable.add("x")
>>> mutable
{1, 2, 'x'}
```

Lists and sets are mutable, as are dictionaries and other mapping types. Strings and tuples are not mutable. Attempting to modify an element of an immutable object will raise a `TypeError`.

Conclusion

Python works differently from languages that support passing arguments by reference or by value. Function arguments become local variables assigned to each value that was passed to the function. But this doesn't prevent you from achieving the same results you'd expect

to the function. But this doesn't prevent you from achieving the same results you'd expect when passing arguments by reference in other languages.

In this tutorial, you learned:

- How Python handles **assigning values to variables**
- How function arguments are **passed by assignment** in Python
- Why **returning values** is a best practice for replicating pass by reference
- How to use **attributes**, **dictionaries**, and **lists** as alternative best practices

You also learned some additional best practices for replicating pass-by-reference constructs in Python. You can use this knowledge to implement patterns that have traditionally required support for passing by reference.

To continue your Python journey, I encourage you to dive deeper into some of the related topics that you've encountered here, such as mutability, assignment expressions, and Python namespaces and scope.

Stay curious, and see you next time!

Mark as Completed

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Pass by Reference in Python: Best Practices**

About **Marius Mogyorosi**

Marius is a tinkerer who loves using Python for creative projects within and beyond the software security field.

» More about Marius

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 David

 Geir Arne

 Joanna

 Jacob

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices intermediate python

Recommended Video Course: Pass by Reference in Python: Best Practices