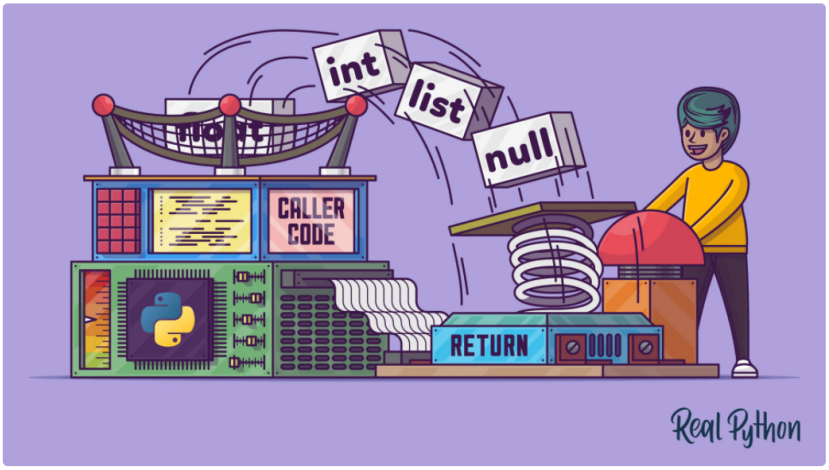

The Python return Statement: Usage and Best Practices

by Leodanis Pozo Ramos Sep 28, 2020 3 Comments basics best-practices python

Mark as Completed

Table of Contents

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: **Using the Python return Statement Effectively**

The Python return statement is a key component of functions and methods. You can use the return statement to make your functions send Python objects back to the caller code. These

Table of Contents

- Getting Started With Python Functions
- Understanding the Python return Statement
- Returning vs Printing
- Returning Multiple Values
- Using the Python return Statement: Best Practices
- Returning Functions: Closures
- Taking and Returning Functions: Decorators
- Returning User-Defined Objects: The Factory Pattern
- Using return in try … finally Blocks
- Using return in Generator Functions
- Conclusion

Mark as Completed

return statement to make your functions send Python objects back to the caller code. These objects are known as the function's **return value**. You can use them to perform further computation in your programs.

Using the `return` statement effectively is a core skill if you want to code custom functions that are Pythonic and robust.

In this tutorial, you'll learn:

- How to use the **Python `return` statement** in your functions
- How to return **single** or **multiple values** from your functions
- What **best practices** to observe when using `return` statements

With this knowledge, you'll be able to write more readable, maintainable, and concise functions in Python. If you're totally new to Python functions, then you can check out Defining Your Own Python Function before diving into this tutorial.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Getting Started With Python Functions

Most programming languages allow you to assign a name to a code block that performs a concrete computation. These named code blocks can be reused quickly because you can use their name to call them from different places in your code.

Programmers call these named code blocks **subroutines**, **routines**, **procedures**, or **functions** depending on the language they use. In some languages, there's a clear difference between a routine or procedure and a function.

Sometimes that difference is so strong that you need to use a specific keyword to define a procedure or subroutine and another keyword to define a function. For example the Visual Basic programming language uses `Sub` and `Function` to differentiate between the two.

In general, a **procedure** is a named code block that performs a set of actions without computing a final value or result. On the other hand, a **function** is a named code block that performs some actions with the purpose of computing a final value or result, which is then sent back to the caller code. Both procedures and functions can act upon a set of **input values**, commonly known as **arguments**.

In Python, these kinds of named code blocks are known as functions because they always send a value back to the caller. The Python documentation defines a function as follows:

> A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. (Source)

Even though the official documentation states that a function "returns some value to the caller," you'll soon see that functions can return any Python object to the caller code.

In general, a function **takes** arguments (if any), **performs** some operations, and **returns** a value (or object). The value that a function returns to the caller is generally known as the function's **return value**. All Python functions have a return value, either explicit or implicit. You'll cover the difference between explicit and implicit return values later in this tutorial.

To write a Python function, you need a **header** that starts with the `def` keyword, followed by the name of the function, an optional list of comma-separated arguments inside a required pair of parentheses, and a final colon.

The second component of a function is its **code block**, or **body**. Python defines code blocks using indentation instead of brackets, `begin` and `end` keywords, and so on. So, to define a function in Python you can use the following syntax:

```
Python
def function_name(arg1, arg2,..., argN):
    # Function's code goes here
```

```
    pass
```

When you're coding a Python function, you need to define a header with the `def` keyword, the name of the function, and a list of arguments in parentheses. Note that the list of arguments is optional, but the parentheses are syntactically required. Then you need to define the function's code block, which will begin one level of indentation to the right.

In the above example, you use a `pass` statement. This kind of statement is useful when you need a placeholder statement in your code to make it syntactically correct, but you don't need to perform any action. `pass` statements are also known as the **null operation** because they don't perform any action.

> **Note:** The full syntax to define functions and their arguments is beyond the scope of this tutorial. For an in-depth resource on this topic, check out Defining Your Own Python Function.

To use a function, you need to call it. A function call consists of the function's name followed by the function's arguments in parentheses:

Python

```
function_name(arg1, arg2, ..., argN)
```

You'll need to pass arguments to a function call only if the function requires them. The parentheses, on the other hand, are always required in a function call. If you forget them, then you won't be calling the function but referencing it as a function object.

To make your functions return a value, you need to use the Python `return` statement. That's what you'll cover from this point on.

Understanding the Python `return` Statement

The **Python `return` statement** is a special statement that you can use inside a function or method to send the function's result back to the caller. A `return` statement consists of the `return` keyword followed by an optional **return value**.

The return value of a Python function can be any Python object. Everything in Python is an object. So, your functions can return numeric values (`int`, `float`, and `complex` values), collections and sequences of objects (`list`, `tuple`, `dictionary`, or `set` objects), user-defined objects, classes, functions, and even modules or packages.

You can omit the return value of a function and use a bare `return` without a return value. You can also omit the entire `return` statement. In both cases, the return value will be `None`.

In the next two sections, you'll cover the basics of how the `return` statement works and how you can use it to return the function's result back to the caller code.

Explicit `return` Statements

An **explicit `return` statement** immediately terminates a function execution and sends the return value back to the caller code. To add an explicit `return` statement to a Python function, you need to use `return` followed by an optional return value:

Python >>>

```
>>> def return_42():
...     return 42  # An explicit return statement
...

>>> return_42()  # The caller code gets 42
42
```

When you define `return_42()`, you add an explicit `return` statement (`return 42`) at the end of the function's code block. `42` is the explicit return value of `return_42()`. This means that any time you call `return_42()`, the function will send `42` back to the caller.

> **Note:** You can use explicit `return` statements with or without a return value. If you build a `return` statement without specifying a return value, then you'll be implicitly returning `None`.

If you define a function with an explicit `return` statement that has an explicit return value, then you can use that return value in any expression:

Python >>>

```
>>> num = return_42()
>>> num
42

>>> return_42() * 2
84

>>> return_42() + 5
47
```

Since `return_42()` returns a numeric value, you can use that value in a math expression or any other kind of expression in which the value has a logical or coherent meaning. This is how a caller code can take advantage of a function's return value.

Note that you can use a `return` statement only inside a function or method definition. If you use it anywhere else, then you'll get a `SyntaxError`:

Python >>>

```
>>> return 42
  File "<stdin>", line 1
SyntaxError: 'return' outside function
```

When you use `return` outside a function or method, you get a `SyntaxError` telling you that the statement can't be used outside a function.

> **Note:** Regular methods, class methods, and static methods are just functions within the context of Python classes. So, all the `return` statement concepts that you'll cover apply to them as well.

You can use any Python object as a return value. Since everything in Python is an object, you can return strings, lists, tuples, dictionaries, functions, classes, instances, user-defined objects, and even modules or packages.

For example, say you need to write a function that takes a list of integers and returns a list containing only the even numbers in the original list. Here's a way of coding this function:

Python >>>

```
>>> def get_even(numbers):
...     even_nums = [num for num in numbers if not num % 2]
...     return even_nums
...

>>> get_even([1, 2, 3, 4, 5, 6])
[2, 4, 6]
```

`get_even()` uses a list comprehension to create a list that filters out the odd numbers in the original `numbers`. Then the function returns the resulting list, which contains only even numbers.

A common practice is to use the result of an expression as a return value in a `return` statement. To apply this idea, you can rewrite `get_even()` as follows:

Python >>>

```
>>> def get_even(numbers):
...     return [num for num in numbers if not num % 2]
...

>>> get_even([1, 2, 3, 4, 5, 6])
[2, 4, 6]
```

The list comprehension gets evaluated and then the function returns with the resulting list. Note that you can only use expressions in a `return` statement. Expressions are different from statements like conditionals or loops.

> **Note:** Even though `list` comprehensions are built using `for` and (optionally) `if` keywords, they're considered expressions rather than statements. That's why you can use them in a `return` statement.

For a further example, say you need to calculate the mean of a sample of numeric values. To do that, you need to divide the sum of the values by the number of values. Here's an example that uses the built-in functions `sum()` and `len()`:

Python >>>

```
>>> def mean(sample):
...     return sum(sample) / len(sample)
...

>>> mean([1, 2, 3, 4])
2.5
```

In `mean()`, you don't use a local variable to store the result of the calculation. Instead, you use the expression directly as a return value. Python first evaluates the expression `sum(sample) / len(sample)` and then returns the result of the evaluation, which in this case is the value `2.5`.

Implicit `return` Statements

A Python function will always have a return value. There is no notion of procedure or routine in Python. So, if you don't explicitly use a return value in a `return` statement, or if you totally omit the `return` statement, then Python will implicitly return a default value for you. That default return value will always be `None`.

Say you're writing a function that adds `1` to a number `x`, but you forget to supply a `return` statement. In this case, you'll get an **implicit `return` statement** that uses `None` as a return value:

Python >>>

```
>>> def add_one(x):
...     # No return statement at all
...     result = x + 1
...

>>> value = add_one(5)
>>> value

>>> print(value)
None
```

If you don't supply an explicit `return` statement with an explicit return value, then Python will supply an implicit `return` statement using `None` as a return value. In the above example, `add_one()` adds `1` to `x` and stores the value in `result` but it doesn't return `result`. That's why you get `value = None` instead of `value = 6`. To fix the problem, you need to either `return result` or directly `return x + 1`.

An example of a function that returns `None` is `print()`. The goal of this function is to print objects to a text stream file, which is normally the standard output (your screen). So, this function doesn't need an explicit `return` statement because it doesn't return anything useful or meaningful:

```
Python >>>
>>> return_value = print("Hello, World")
Hello, World

>>> print(return_value)
None
```

The call to `print()` prints `Hello, World` to the screen. Since this is the purpose of `print()`, the function doesn't need to return anything useful, so you get `None` as a return value.

> **Note:** The Python interpreter doesn't display `None`. So, to show a return value of `None` in an interactive session, you need to explicitly use `print()`.

Regardless of how long and complex your functions are, any function without an explicit `return` statement, or one with a `return` statement without a return value, will return `None`.

Returning vs Printing

If you're working in an interactive session, then you might think that printing a value and returning a value are equivalent operations. Consider the following two functions and their output:

```
Python >>>
>>> def print_greeting():
...     print("Hello, World")
...

>>> print_greeting()
Hello, World

>>> def return_greeting():
...     return "Hello, World"
...

>>> return_greeting()
'Hello, World'
```

Both functions seem to do the same thing. In both cases, you see `Hello, World` printed on your screen. There's only a subtle visible difference—the single quotation marks in the second example. But take a look at what happens if you return another data type, say an `int` object:

```
Python >>>
>>> def print_42():
...     print(42)
...

>>> print_42()
42

>>> def return_42():
...     return 42
...

>>> return_42()
42
```

There's no visible difference now. In both cases, you can see `42` on your screen. That behavior can be confusing if you're just starting with Python. You might think that returning and printing a value are equivalent actions.

Now, suppose you're getting deeper into Python and you're starting to write your first script. You open a text editor and type the following code:

```python
def add(a, b):
    result = a + b
    return result

add(2, 2)
```

`add()` takes two numbers, adds them, and returns the result. On **line 5**, you call `add()` to sum `2` plus `2`. Since you're still learning the difference between returning and printing a value, you might expect your script to print `4` to the screen. However, that's not what happens, and you get nothing on your screen.

Try it out by yourself. Save your script to a file called `adding.py` and run it from your command line as follows:

```shell
$ python3 adding.py
```

If you run `adding.py` from your command line, then you won't see any result on your screen. That's because when you run a script, the return values of the functions that you call in the script don't get printed to the screen like they do in an interactive session.

If you want that your script to show the result of calling `add()` on your screen, then you need to explicitly call `print()`. Check out the following update of `adding.py`:

```python
def add(a, b):
    result = a + b
    return result

print(add(2, 2))
```

Now, when you run `adding.py`, you'll see the number `4` on your screen.

So, if you're working in an interactive session, then Python will show the result of any function call directly to your screen. But if you're writing a script and you want to see a function's return value, then you need to explicitly use `print()`.

Returning Multiple Values

You can use a `return` statement to return multiple values from a function. To do that, you just need to supply several return values separated by commas.

For example, suppose you need to write a function that takes a sample of numeric data and returns a summary of statistical measures. To code that function, you can use the Python standard module `statistics`, which provides several functions for calculating mathematical statistics of numeric data.

Here's a possible implementation of your function:

```python
import statistics as st

def describe(sample):
    return st.mean(sample), st.median(sample), st.mode(sample)
```

In `describe()`, you take advantage of Python's ability to return multiple values in a single `return` statement by returning the mean, median, and mode of the sample at the same time. Note that, to return multiple values, you just need to write them in a comma-separated list in the order you want them returned.

in the order you want them returned.

Once you've coded `describe()`, you can take advantage of a powerful Python feature known as iterable unpacking to unpack the three measures into three separated variables, or you can just store everything in one variable:

```
Python                                                    >>>
>>> sample = [10, 2, 4, 7, 9, 3, 9, 8, 6, 7]
>>> mean, median, mode = describe(sample)

>>> mean
6.5

>>> median
7.0

>>> mode
7

>>> desc = describe(sample)
>>> desc
(6.5, 7.0, 7)

>>> type(desc)
<class 'tuple'>
```

Here, you unpack the three return values of `describe()` into the variables `mean`, `median`, and `mode`. Note that in the last example, you store all the values in a single variable, `desc`, which turns out to be a Python `tuple`.

> **Note:** You can build a Python `tuple` by just assigning several comma-separated values to a single variable. There's no need to use parentheses to create a `tuple`. That's why multiple return values are packed in a `tuple`.

The built-in function `divmod()` is also an example of a function that returns multiple values. The function takes two (non-complex) numbers as arguments and returns two numbers, the quotient of the two input values and the remainder of the division:

```
Python                                                    >>>
>>> divmod(15, 3)
(5, 0)

>>> divmod(8, 3)
(2, 2)
```

The call to `divmod()` returns a tuple containing the quotient and remainder that result from dividing the two non-complex numbers provided as arguments. This is an example of a function with multiple return values.

Using the Python `return` Statement: Best Practices

So far, you've covered the basics of how the Python `return` statement works. You now know how to write functions that return one or multiple values to the caller. Additionally, you've learned that if you don't add an explicit `return` statement with an explicit return value to a given function, then Python will add it for you. That value will be `None`.

In this section, you'll cover several examples that will guide you through a set of good programming practices for effectively using the `return` statement. These practices will help you to write more readable, maintainable, robust, and efficient functions in Python.

Returning `None` Explicitly

Some programmers rely on the implicit `return` statement that Python adds to any function without an explicit one. This can be confusing for developers who come from other programming languages in which a function without a return value is called a **procedure**.

There are situations in which you can add an explicit `return None` to your functions. In other situations, however, you can rely on Python's default behavior:

- If your function performs actions but doesn't have a clear and useful `return` value, then you can omit returning `None` because doing that would just be superfluous and confusing. You can also use a bare `return` without a return value just to make clear your intention of returning from the function.
- If your function has multiple `return` statements and returning `None` is a valid option, then you should consider the explicit use of `return None` instead of relying on the Python's default behavior.

These practices can improve the readability and maintainability of your code by explicitly communicating your intent.

When it comes to returning `None`, you can use one of three possible approaches:

1. Omit the `return` statement and rely on the default behavior of returning `None`.
2. Use a bare `return` without a return value, which also returns `None`.
3. Return `None` explicitly.

Here's how this works in practice:

Python >>>

```
>>> def omit_return_stmt():
...     # Omit the return statement
...     pass
...
>>> print(omit_return_stmt())
None

>>> def bare_return():
...     # Use a bare return
...     return
...
>>> print(bare_return())
None

>>> def return_none_explicitly():
...     # Return None explicitly
...     return None
...
>>> print(return_none_explicitly())
None
```

Whether or not to return `None` explicitly is a personal decision. However, you should consider that in some cases, an explicit `return None` can avoid maintainability problems. This is especially true for developers who come from other programming languages that don't behave like Python does.

Remembering the Return Value

When writing custom functions, you might accidentally forget to return a value from a function. In this case, Python will return `None` for you. This can cause subtle bugs that can be difficult for a beginning Python developer to understand and debug.

You can avoid this problem by writing the `return` statement immediately after the header of the function. Then you can make a second pass to write the function's body. Here's a template that you can use when coding your Python functions:

Python

```
def template_func(args):
    result = 0  # Initialize the return value
    # Your code goes here...
```

```
    return result  # Explicitly return the result
```

If you get used to starting your functions like this, then chances are that you'll no longer miss the `return` statement. With this approach, you can write the body of the function, test it, and rename the variables once you know that the function works.

This practice can increase your productivity and make your functions less error-prone. It can also save you a lot of debugging time.

Avoiding Complex Expressions

As you saw before, it's a common practice to use the result of an expression as a return value in Python functions. If the expression that you're using gets too complex, then this practice can lead to functions that are difficult to understand, debug, and maintain.

For example, if you're doing a complex calculation, then it would be more readable to incrementally calculate the final result using **temporary variables** with meaningful names.

Consider the following function that calculates the variance of a sample of numeric data:

Python >>>

```
>>> def variance(data, ddof=0):
...     mean = sum(data) / len(data)
...     return sum((x - mean) ** 2 for x in data) / (len(data) - ddof)
...

>>> variance([3, 4, 7, 5, 6, 2, 9, 4, 1, 3])
5.24
```

The expression that you use here is quite complex and difficult to understand. It's also difficult to debug because you're performing multiple operations in a single expression. To work around this particular problem, you can take advantage of an incremental development approach that improves the readability of the function.

Take a look at the following alternative implementation of `variance()`:

Python >>>

```
>>> def variance(data, ddof=0):
...     n = len(data)
...     mean = sum(data) / n
...     total_square_dev = sum((x - mean) ** 2 for x in data)
...     return total_square_dev / (n - ddof)
...

>>> variance([3, 4, 7, 5, 6, 2, 9, 4, 1, 3])
5.24
```

In this second implementation of `variance()`, you calculate the variance in several steps. Each step is represented by a temporary variable with a meaningful name.

Temporary variables like `n`, `mean`, and `total_square_dev` are often helpful when it comes to debugging your code. If, for example, something goes wrong with one of them, then you can call `print()` to know what's happening before the `return` statement runs.

In general, you should avoid using complex expressions in your `return` statement. Instead, you can break your code into multiple steps and use temporary variables for each step. Using temporary variables can make your code easier to debug, understand, and maintain.

Returning Values vs Modifying Globals

Functions that don't have an explicit `return` statement with a meaningful return value often preform actions that have side effects. A **side effect** can be, for example, printing something to the screen, modifying a global variable, updating the state of an object, writing some text to a file, and so on.

Modifying global variables is generally considered a bad programming practice. Just like programs with complex expressions, programs that modify global variables can be difficult

to debug, understand, and maintain.

When you modify a global variables, you're potentially affecting all the functions, classes, objects, and any other parts of your programs that rely on that global variable.

To understand a program that modifies global variables, you need to be aware of all the parts of the program that can see, access, and change those variables. So, good practice recommends writing **self-contained functions** that take some arguments and return a useful value (or values) without causing any side effect on global variables.

Additionally, functions with an explicit `return` statement that return a meaningful value are easier to test than functions that modify or update global variables.

The following example show a function that changes a global variable. The function uses the `global` statement, which is also considered a bad programming practice in Python:

Python >>>

```
>>> counter = 0

>>> def increment():
...     global counter
...     counter += 1
...

>>> increment()
>>> counter
1
```

In this example, you first create a global variable, `counter`, with an initial value of `0`. Inside `increment()`, you use a `global` statement to tell the function that you want to modify a global variable. The last statement increments `counter` by `1`.

The result of calling `increment()` will depend on the initial value of `counter`. Different initial values for `counter` will generate different results, so the function's result can't be controlled by the function itself.

To avoid this kind of behavior, you can write a self-contained `increment()` that takes arguments and returns a coherent value that depends only on the input arguments:

Python >>>

```
>>> counter = 0

>>> def increment(var):
...     return var + 1
...

>>> increment(counter)
1

>>> counter
0

>>> # Explicitly assign a new value to counter
>>> counter = increment(counter)
>>> counter
1
```

Now the result of calling `increment()` depends only on the input arguments rather than on the initial value of `counter`. This makes the function more robust and easier to test.

> **Note:** For a better understanding of how to test your Python code, check out Test-Driven Development With PyTest.

Additionally, when you need to update `counter`, you can do so explicitly with a call to `increment()`. This way, you'll have more control over what's happening with `counter` throughout your code.

In general, it's a good practice to avoid functions that modify global variables. If possible, try to write **self-contained functions** with an explicit `return` statement that returns a coherent

and meaningful value.

Using `return` With Conditionals

Python functions are not restricted to having a single `return` statement. If a given function has more than one `return` statement, then the first one encountered will determine the end of the function's execution and also its return value.

A common way of writing functions with multiple `return` statements is to use conditional statements that allow you to provide different `return` statements depending on the result of evaluating some conditions.

Suppose you need to code a function that takes a number and returns its absolute value. If the number is greater than `0`, then you'll return the same number. If the number is less than `0`, then you'll return its opposite, or non-negative value.

Here's a possible implementation for this function:

Python >>>

```
>>> def my_abs(number):
...     if number > 0:
...         return number
...     elif number < 0:
...         return -number
...

>>> my_abs(-15)
15

>>> my_abs(15)
15
```

`my_abs()` has two explicit `return` statements, each of them wrapped in its own `if` statement. It also has an implicit `return` statement. If `number` happens to be `0`, then neither condition is true, and the function ends without hitting any explicit `return` statement. When this happens, you automatically get `None`.

Take a look at the following call to `my_abs()` using `0` as an argument:

Python >>>

```
>>> print(my_abs(0))
None
```

When you call `my_abs()` using `0` as an argument, you get `None` as a result. That's because the flow of execution gets to the end of the function without reaching any explicit `return` statement. Unfortunately, the absolute value of `0` is `0`, not `None`.

To fix this problem, you can add a third `return` statement, either in a new `elif` clause or in a final `else` clause:

Python >>>

```
>>> def my_abs(number):
...     if number > 0:
...         return number
...     elif number < 0:
...         return -number
...     else:
...         return 0
...

>>> my_abs(0)
0

>>> my_abs(-15)
```

```
15

>>> my_abs(15)
15
```

Now, `my_abs()` checks every possible condition, `number > 0`, `number < 0`, and `number == 0`. The purpose of this example is to show that when you're using conditional statements to provide multiple `return` statements, you need to make sure that every possible option gets its own `return` statement. Otherwise, your function will have a hidden bug.

Finally, you can implement `my_abs()` in a more concise, efficient, and Pythonic way using a single `if` statement:

Python >>>

```
>>> def my_abs(number):
...     if number < 0:
...         return -number
...     return number
...

>>> my_abs(0)
0

>>> my_abs(-15)
15

>>> my_abs(15)
15
```

In this case, your function hits the first `return` statement if `number < 0`. In all other cases, whether `number > 0` or `number == 0`, it hits the second `return` statement. With this new implementation, your function looks a lot better. It's more readable, concise, and efficient.

> **Note:** There's a convenient built-in Python function called `abs()` for computing the absolute value of a number. The function in the above example is intended only to illustrate the point under discussion.

If you're using `if` statements to provide several `return` statements, then you don't need an `else` clause to cover the last condition. Just add a `return` statement at the end of the function's code block and at the first level of indentation.

Returning `True` or `False`

Another common use case for the combination of `if` and `return` statements is when you're coding a predicate or Boolean-valued function. This kind of function returns either `True` or `False` according to a given condition.

For example, say you need to write a function that takes two integers, `a` and `b`, and returns `True` if `a` is divisible by `b`. Otherwise, the function should return `False`. Here's a possible implementation:

Python >>>

```
>>> def is_divisible(a, b):
...     if not a % b:
...         return True
...     return False
...

>>> is_divisible(4, 2)
True

>>> is_divisible(7, 4)
False
```

`is_divisible()` returns `True` if the remainder of dividing `a` by `b` is equal to `0`. Otherwise, it returns `False`. Note that in Python, a `0` value is falsy, so you need to use the `not` operator to negate the truth value of the condition.

Sometimes you'll write predicate functions that involve operators like the following:

- The comparison operators `==`, `!=`, `>`, `>=`, `<`, and `<=`
- The membership operator `in`
- The identity operator `is`
- The Boolean operator `not`

In these cases, you can directly use a Boolean expression in your `return` statement. This is possible because these operators return either `True` or `False`. Following this idea, here's a new implementation of `is_divisible()`:

Python >>>

```
>>> def is_divisible(a, b):
...     return not a % b
...

>>> is_divisible(4, 2)
True

>>> is_divisible(7, 4)
False
```

If `a` is divisible by `b`, then `a % b` returns `0`, which is falsy in Python. So, to return `True`, you need to use the `not` operator.

> **Note:** Python follows a set of rules to determine the truth value of an object.
>
> For example, the following objects are considered falsy:
>
> - Constants like `None` and `False`
> - Numeric types with a zero value like `0`, `0.0`, `0j`, `Decimal(0)`, and `Fraction(0, 1)`
> - Empty sequences and collections like `""`, `()`, `[]`, `{}`, `set()`, and `range(0)`
> - Objects that implement `__bool__()` with a return value of `False` or `__len__()` with a return value of `0`
>
> Any other object will be considered truthy.

On the other hand, if you try to use conditions that involve Boolean operators like `or` and `and` in the way you saw before, then your predicate functions won't work correctly. That's because these operators behave differently. They return one of the operands in the condition rather than `True` or `False`:

Python >>>

```
>>> 0 and 1
0
>>> 1 and 2
2

>>> 1 or 2
1
>>> 0 or 1
1
```

In general, `and` returns the first false operand or the last operand. On the other hand, `or` returns the first true operand or the last operand. So, to write a predicate that involves one of these operators, you'll need to use an explicit `if` statement or a call to the built-in function `bool()`.

Suppose you want to write a predicate function that takes two values and returns `True` if both are true and `False` otherwise. Here's your first approach to this function:

Python >>>

```
>>> def both_true(a, b):
...     return a and b
...
>>> both_true(1, 2)
2
```

Since `and` returns operands instead of `True` or `False`, your function doesn't work correctly. There are at least three possibilities for fixing this problem:

1. An explicit `if` statement
2. A conditional expression (ternary operator)
3. The built-in Python function `bool()`

If you use the first approach, then you can write `both_true()` as follows:

```
Python >>>
>>> def both_true(a, b):
...     if a and b:
...         return True
...     return False
...

>>> both_true(1, 2)
True

>>> both_true(1, 0)
False
```

The `if` statement checks if `a` and `b` are both truthy. If so, then `both_true()` returns `True`. Otherwise, it returns `False`.

If, on the other hand, you use a Python conditional expression or ternary operator, then you can write your predicate function as follows:

```
Python >>>
>>> def both_true(a, b):
...     return True if a and b else False
...
>>> both_true(1, 2)
True

>>> both_true(1, 0)
False
```

Here, you use a conditional expression to provide a return value for `both_true()`. The conditional expression is evaluated to `True` if both `a` and `b` are truthy. Otherwise, the final result is `False`.

Finally, if you use `bool()`, then you can code `both_true()` as follows:

```
Python >>>
>>> def both_true(a, b):
...     return bool(a and b)
...

>>> both_true(1, 2)
True

>>> both_true(1, 0)
False
```

`bool()` returns `True` if `a` and `b` are true and `False` otherwise. It's up to you what approach to use for solving this problem. However, the second solution seems more readable. What do you think?

Short-Circuiting Loops

A `return` statement inside a loop performs some kind of **short-circuit**. It breaks the loop

execution and makes the function return immediately. To better understand this behavior, you can write a function that emulates `any()`. This built-in function takes an iterable and returns `True` if at least one of its items is truthy.

To emulate `any()`, you can code a function like the following:

```
Python >>>
>>> def my_any(iterable):
...     for item in iterable:
...         if item:
...             # Short-circuit
...             return True
...     return False

>>> my_any([0, 0, 1, 0, 0])
True

>>> my_any([0, 0, 0, 0, 0])
False
```

If any `item` in `iterable` is true, then the flow of execution enters in the `if` block. The `return` statement breaks the loop and returns immediately with a return value of `True`. If no value in `iterable` is true, then `my_any()` returns `False`.

This function implements a **short-circuit evaluation**. For example, suppose that you pass an iterable that contains a million items. If the first item in that iterable happens to be true, then the loop runs only one time rather than a million times. This can save you a lot of processing time when running your code.

It's important to note that to use a `return` statement inside a loop, you need to wrap the statement in an `if` statement. Otherwise, the loop will always break in its first iteration.

Recognizing Dead Code

As soon as a function hits a `return` statement, it terminates without executing any subsequent code. Consequently, the code that appears after the function's `return` statement is commonly called **dead code**. The Python interpreter totally ignores dead code when running your functions. So, having that kind of code in a function is useless and confusing.

Consider the following function, which adds code after its `return` statement:

```
Python >>>
>>> def dead_code():
...     return 42
...     # Dead code
...     print("Hello, World")
...

>>> dead_code()
42
```

The statement `print("Hello, World")` in this example will never execute because that statement appears after the function's `return` statement. Identifying dead code and removing it is a good practice that you can apply to write better functions.

It's worth noting that if you're using conditional statements to provide multiple `return` statements, then you can have code after a `return` statement that won't be dead as long as it's outside the `if` statement:

```
Python >>>
>>> def no_dead_code(condition):
...     if condition:
...         return 42
...     print("Hello, World")
...

>>> no_dead_code(True)
42
>>> no_dead_code(False)
```

```
Hello, World
```

Even though the call to `print()` is after a `return` statement, it's not dead code. When `condition` is evaluated to `False`, the `print()` call is run and you get `Hello, World` printed to your screen.

Returning Multiple Named-Objects

When you're writing a function that returns multiple values in a single `return` statement, you can consider using a `collections.namedtuple` object to make your functions more readable. `namedtuple` is a collection class that returns a subclass of `tuple` that has fields or attributes. You can access those attributes using dot notation or an indexing operation.

The initializer of `namedtuple` takes several arguments. However, to start using `namedtuple` in your code, you just need to know about the first two:

1. **`typename`** holds the name of the tuple-like class that you're creating. It needs to be a string.
2. **`field_names`** holds the names of the fields or attributes of the tuple-like class. It can be a sequence of strings such as `["x", "y"]` or a single string with each name separated by whitespace or commas, such as `"x y"` or `"x, y"`.

Using a `namedtuple` when you need to return multiple values can make your functions significantly more readable without too much effort. Consider the following update of `describe()` using a `namedtuple` as a return value:

Python

```python
import statistics as st
from collections import namedtuple

def describe(sample):
    Desc = namedtuple("Desc", ["mean", "median", "mode"])
    return Desc(
        st.mean(sample),
        st.median(sample),
        st.mode(sample),
    )
```

Inside `describe()`, you create a `namedtuple` called `Desc`. This object can have named attributes that you can access by using dot notation or by using an indexing operation. In this example, those attributes are `"mean"`, `"median"`, and `"mode"`.

You can create a `Desc` object and use it as a return value. To do that, you need to instantiate `Desc` like you'd do with any Python class. Note that you need to supply a concrete value for each named attribute, just like you did in your `return` statement.

Here's how `describe()` works now:

Python >>>

```
>>> sample = [10, 2, 4, 7, 9, 3, 9, 8, 6, 7]
>>> stat_desc = describe(sample)

>>> stat_desc
Desc(mean=5.7, median=6.0, mode=6)

>>> # Get the mean by its attribute name
>>> stat_desc.mean
5.7

>>> # Get the median by its index
>>> stat_desc[1]
6.0

>>> # Unpack the values into three variables
>>> mean, median, mode = describe(sample)

>>> mean
5.7

>>> mode
```

```
6
```

When you call `describe()` with a sample of numeric data, you get a `namedtuple` object containing the mean, median, and mode of the sample. Note that you can access each element of the tuple by using either dot notation or an indexing operation.

Finally, you can also use an iterable unpacking operation to store each value in its own independent variable.

Returning Functions: Closures

In Python, functions are first-class objects. A first-class object is an object that can be assigned to a variable, passed as an argument to a function, or used as a return value in a function. So, you can use a function object as a return value in any `return` statement.

A function that takes a function as an argument, returns a function as a result, or both is a higher-order function. A closure factory function is a common example of a higher-order function in Python. This kind of function takes some arguments and returns an inner function. The inner function is commonly known as a **closure**.

A closure carries information about its enclosing execution scope. This provides a way to retain state information between function calls. Closure factory functions are useful when you need to write code based on the concept of lazy or delayed evaluation.

Suppose you need to write a helper function that takes a number and returns the result of multiplying that number by a given factor. You can code that function as follows:

Python

```
def by_factor(factor, number):
    return factor * number
```

`by_factor()` takes `factor` and `number` as arguments and returns their product. Since `factor` rarely changes in your application, you find it annoying to supply the same factor in every function call. So, you need a way to retain the state or value of `factor` between calls to `by_factor()` and change it only when needed. To retain the current value of `factor` between calls, you can use a closure.

The following implementation of `by_factor()` uses a closure to retain the value of `factor` between calls:

Python >>>

```
>>> def by_factor(factor):
...     def multiply(number):
...         return factor * number
...     return multiply
...

>>> double = by_factor(2)
>>> double(3)
6
>>> double(4)
8

>>> triple = by_factor(3)
>>> triple(3)
9
>>> triple(4)
12
```

Inside `by_factor()`, you define an inner function called `multiply()` and return it without calling it. The function object you return is a closure that retains information about the state of `factor`. In other words, it remembers the value of `factor` between calls. That's why `double`

remembers that `factor` was equal to `2` and `triple` remembers that `factor` was equal to `3`.

Note that you can freely reuse `double` and `triple` because they don't forget their respective state information.

You can also use a `lambda` function to create closures. Sometimes the use of a `lambda` function can make your closure factory more concise. Here's an alternative implementation of `by_factor()` using a `lambda` function:

```
Python                                                    >>>
>>> def by_factor(factor):
...     return lambda number: factor * number
...

>>> double = by_factor(2)
>>> double(3)
6
>>> double(4)
8
```

This implementation works just like the original example. In this case, the use of a `lambda` function provides a quick and concise way to code `by_factor()`.

Taking and Returning Functions: Decorators

Another way of using the `return` statement for returning function objects is to write decorator functions. A **decorator function** takes a function object as an argument and returns a function object. The decorator processes the decorated function in some way and returns it or replaces it with another function or callable object.

Decorators are useful when you need to add extra logic to existing functions without modifying them. For example, you can code a decorator to log function calls, validate the arguments to a function, measure the execution time of a given function, and so on.

The following example shows a decorator function that you can use to get an idea of the execution time of a given Python function:

```
Python                                                    >>>
>>> import time

>>> def my_timer(func):
...     def _timer(*args, **kwargs):
...         start = time.time()
...         result = func(*args, **kwargs)
...         end = time.time()
...         print(f"Execution time: {end - start}")
...         return result
...     return _timer
...

>>> @my_timer
... def delayed_mean(sample):
...     time.sleep(1)
...     return sum(sample) / len(sample)
...

>>> delayed_mean([10, 2, 4, 7, 9, 3, 9, 8, 6, 7])
Execution time: 1.0011096000671387
6.5
```

The syntax `@my_timer` above the header of `delayed_mean()` is equivalent to the expression `delayed_mean = my_timer(delayed_mean)`. In this case, you can say that `my_timer()` is decorating `delayed_mean()`.

Python runs decorator functions as soon as you import or run a module or a script. So, when you call `delayed_mean()`, you're really calling the return value of `my_timer()`, which is the function object `_timer`. The call to the decorated `delayed_mean()` will return the mean of the sample and will also measure the execution time of the original `delayed_mean()`.

In this case, you use `time()` to measure the execution time inside the decorator. `time()` lives

in a module called `time` that provides a set of time-related functions. `time()` returns the time in seconds since the epoch as a floating-point number. The difference between the time before and after the call to `delayed_mean()` will give you an idea of the function's execution time.

> **Note:** In `delayed_mean()`, you use the function `time.sleep()`, which suspends the execution of the calling code for a given number of seconds. For a better understanding on how to use `sleep()`, check out Python sleep(): How to Add Time Delays to Your Code.

Other common examples of decorators in Python are `classmethod()`, `staticmethod()`, and `property()`. If you want to dive deeper into Python decorators, then take a look at Primer on Python Decorators. You can also check out Python Decorators 101.

Returning User-Defined Objects: The Factory Pattern

The Python `return` statement can also return user-defined objects. In other words, you can use your own custom objects as a return value in a function. A common use case for this capability is the factory pattern.

The factory pattern defines an interface for creating objects on the fly in response to conditions that you can't predict when you're writing a program. You can implement a factory of user-defined objects using a function that takes some initialization arguments and returns different objects according to the concrete input.

Say you're writing a painting application. You need to create different shapes on the fly in response to your user's choices. Your program will have squares, circles, rectangles, and so on. To create those shapes on the fly, you first need to create the shape classes that you're going to use:

Python

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    # Class implementation...

class Square:
    def __init__(self, side):
        self.side = side
    # Class implementation...
```

Once you have a class for each shape, you can write a function that takes the name of the shape as a string and an optional list of arguments (`*args`) and keyword arguments (`**kwargs`) to create and initialize shapes on the fly:

Python

```
def shape_factory(shape_name, *args, **kwargs):
    shapes = {"circle": Circle, "square": Square}
    return shapes[shape_name](*args, **kwargs)
```

This function creates an instance of the concrete shape and returns it to the caller. Now you can use `shape_factory()` to create objects of different shapes in response to the needs of your users:

Python >>>

```
>>> circle = shape_factory("circle", radius=20)
>>> type(circle)
<class '__main__.Circle'>
```

```
>>> circle.radius
20

>>> square = shape_factory("square", side=10)
>>> type(square)
<class '__main__.Square'>
>>> square.side
10
```

If you call `shape_factory()` with the name of the required shape as a string, then you get a new instance of the shape that matches the `shape_name` you've just passed to the factory.

Using `return` in `try ... finally` Blocks

When you use a `return` statement inside a `try` statement with a `finally` clause, that `finally` clause is always executed before the `return` statement. This ensures that the code in the `finally` clause will always run. Check out the following example:

Python >>>

```
>>> def func(value):
...     try:
...         return float(value)
...     except ValueError:
...         return str(value)
...     finally:
...         print("Run this before returning")
...

>>> func(9)
Run this before returning
9.0

>>> func("one")
Run this before returning
'one'
```

When you call `func()`, you get `value` converted to a floating-point number or a string object. Before doing that, your function runs the `finally` clause and prints a message to your screen. Whatever code you add to the `finally` clause will be executed before the function runs its `return` statement.

Using `return` in Generator Functions

A Python function with a `yield` statement in its body is a **generator function**. When you call a generator function, it returns a generator iterator. So, you can say that a generator function is a **generator factory**.

You can use a `return` statement inside a generator function to indicate that the generator is done. The `return` statement will make the generator raise a `StopIteration`. The return value will be passed as an argument to the initializer of `StopIteration` and will be assigned to its `.value` attribute.

Here's a generator that yields `1` and `2` on demand and then returns `3`:

Python >>>

```
>>> def gen():
...     yield 1
...     yield 2
...     return 3
...

>>> g = gen()
>>> g
<generator object gen at 0x7f4ff4853c10>

>>> next(g)
1
>>> next(g)
2
```

```
>>> next(g)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    next(g)
StopIteration: 3
```

`gen()` returns a generator object that yields `1` and `2` on demand. To retrieve each number form the generator object, you can use `next()`, which is a built-in function that retrieves the next item from a Python generator.

The first two calls to `next()` retrieve `1` and `2`, respectively. In the third call, the generator is exhausted, and you get a `StopIteration`. Note that the return value of the generator function (`3`) becomes the `.value` attribute of the `StopIteration` object.

Conclusion

The Python `return` statement allows you to send any Python object from your custom functions back to the caller code. This statement is a fundamental part of any Python function or method. If you master how to use it, then you'll be ready to code robust functions.

In this tutorial, you've learned how to:

- Effectively use the **Python `return` statement** in your **functions**
- Return **single** or **multiple values** from your functions to the caller code
- Apply **best practices** when using the `return` statement

Additionally, you've learned some more advanced use cases for the `return` statement, like how to code a closure factory function and a decorator function. With this knowledge, you'll be able to write more Pythonic, robust, and maintainable functions in Python.

About **Leodanis Pozo Ramos**

Leodanis is an industrial engineer who loves Python and

software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

» More about Leodanis

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

 Bryan

 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

Keep Learning

Related Tutorial Categories: basics best-practices python

Recommended Video Course: Using the Python return Statement Effectively